AN OPEN ARCHITECTURE FOR DISTRIBUTED IP TRAFFIC ANALYSIS

DOCTORAL THESIS

FOR THE DEGREE OF A DOCTOR OF INFORMATICS

AT THE FACULTY OF ECONOMICS,
BUSINESS ADMINISTRATION AND
INFORMATION TECHOLOGY
OF THE UNIVERSITY OF ZURICH

by

CRISTIAN MORARIU

from Romania

Accepted on the recommendation of

PROF. DR. B. STILLER
PROF. DR. A. PRAS

2010

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, December, 2010

The Vice Dean of the Academic Program in Informatics: Prof. Dr. H. C. Gall

Abstract

Since the first days of the Internet the IP (Internet Protocol) traffic carried by network operators increased year after year. This was mainly caused by a continuous growth in the number of users having Internet access, combined with an increase in services that those users have access to using an IP infrastructure. Traditional telecommunication services that had their dedicated infrastructures (such as telephony, television) are in a process of gradually switching to IP. Two additional causes, which have lead to the increase of network traffic are a constant need of users to have access to higher quality services, and, the pervasiveness of modern mobile devices, which allow users to be connected to the Internet anytime and from almost anywhere. Different studies of the evolution of Internet traffic show that on average, during the last decade, Internet traffic increased between 50% to 100% every year, depending on the markets where those studies have been made [16], [69], [80], [100]. Looking at the behavior of Internet users during recent years, and considering Internet services that are still very new, or expected to be available soon, leads to estimations [18], [19] that the trend in traffic increase will continue at the same rate until 2013 and most probably beyond. By 2012 the total Internet traffic carried by Internet providers is estimated to be about 75 times higher than the total traffic carried in 2002 [19].

Besides the impact that such a traffic increase has on the routing and switching infrastructure of network operators also the network monitoring and management mechanisms need to be changed in order to address more traffic. Traffic metering and analysis mechanisms will still be important in the hands of network operators, as these are the basis for many network man-

agement operations, such as network monitoring, planning, intrusion detection, accounting, charging, or billing.

One important problem, which was observed in today's IP traffic metering and analysis mechanisms and which motivates this thesis, is the use of a central element which collects all traffic metering data and which performs traffic analysis tasks on these data. The use of a central element is required by traffic analysis applications which have to correlate different pieces of metered data. High packet rates only allow for a limited number of operations to be made on a packet, before the next one arrives at the metering device. Similarly, as the IP metering data collected by large operators in a single day is in the orders of tens or hundreds of Gigabytes, traffic analysis applications take a long time to analyze that traffic.

The contribution of this research is a set of building blocks for distributed traffic analysis. As traditional centralized approaches for traffic metering and analysis cannot scale with the traffic increase, this thesis proposes a distributed traffic metering and analysis model. A generic architecture for the distribution of captured IP metering data (DITA), which includes a framework for enabling distributed traffic analysis, is presented and evaluated. In addition, the thesis also proposes two metering mechanisms, as instances of the distributed metering process, which address problems of existing solution. The first metering mechanism enables a software-based traffic monitoring application, such as Snort [90], or nTop [79], to run in parallel on several machines in order to increase the number of packets that can be inspected every second. The second metering mechanism allows for the identification of the user and application which sent (or received) a particular IP packet, not just the network device which they used. This tool supports accounting or intrusion detection systems for multi-user operating systems.

The evaluation of these newly proposed mechanisms show that a distribution of traffic metering and traffic analysis is feasible and may be the basis of future scalable IP traffic metering and analysis infrastructures.

Zusammenfassung

Schon seit Anbeginn des Internet wächst von Jahr zu Jahr der IP (Internet Protokoll) Verkehr, den Netzwerkbetreiber übertragen. Dieser Zuwachs ist hauptsächlich durch die kontinuierlich steigende Anzahl von Benutzern mit Zugang zum Internet und mit einem grösser werdenden Angebot an auf IP-Infrastruktur angebotenen Diensten begründet, die den Benutzern zur Verfügung stehen. Traditionelle Fernmeldedienste wie Telefonie oder Fernsehen, die bis anhin mit Hilfe eigener Infrastruktur erbracht wurden, wechseln schrittweise auf IP. Hinzu kommen zwei zusätzliche Ursachen für ansteigenden Netzwerkverkehr, nämlich der beständige Bedarf der Benutzer nach Diensten in erhöhter Qualität und die Allgegenwärtigkeit moderner Mobilgeräte, die Benutzern jederzeit eine Verbindung zum Internet von beinahe überall aus ermöglichen. Verschiedene Untersuchungen zur Entwicklung des Internetverkehrs zeigen für die letzten zehn Jahre in Abhängigkeit desjenigen Marktes, in dem die jeweilige Untersuchung durchgeführt wurde, eine durchschnittliche jährliche Zuwachsrate von 50 bis 100 Prozent [16], [69], [80], [100]. Das in den letzten Jahren zu beobachtende Benutzerverhalten sowie teils noch sehr junge oder sich erst abzeichnende Dienste erlauben Schätzungen [19], denen ein kontinuierlicher Verkehrszuwachstrend in demselben Umfang bis 2013 und höchstwahrscheinlich auch darüber hinaus zugrunde liegen. Für das Jahr 2012 wird das durch Internetanbieter aufgebrachte Verkehrsvolumen insgesamt rund 75 mal höher veranschlagt als der entsprechende Wert für das Jahr 2002 [19].

Neben den mit einem solchen Verkehrszuwachs verbundenen Auswirkungen auf die Wegewahl- und Vermittlungsinfrastruktur eines Netzbetreibers

werden Anpassungen in der Netzwerküberwachung und in den eingesetzten Mechanismen des Netzwerk-Managements erforderlich, um dem Mehrverkehr begegnen zu können. Mechanismen für die Durchführung von Verkehrsmessungen und deren Auswertung werden weiterhin wichtig für Netzwerkbetreiber bleiben, da sie die Grundlage für viele Aufgaben des Netzwerk-Managements bilden, so zum Beispiel für die Netzwerküberwachung, die Netzwerkplanung, das Erkennen von Angriffsversuchen, das Abrechnen, Verrechnen oder die Rechnungsstellung.

Ein zentrales Problem der heute eingesetzten Mechanismen zum Messen und Auswerten von IP-Verkehr stellt die Motivation für die vorliegende Dissertation dar. Es besteht im Einsatz eines zentralen Elements, das alle Verkehrsmessungsdaten sammelt und darauf die Verkehrsanalyse durchführt. Ein zentrales Element ist erforderlich für diejenigen Anwendungen der Verkehrsauswertung, die unterschiedliche Bereiche der Messdaten korrelieren müssen. Dabei gilt es zu beachten, dass hohe Paketraten nur eine begrenzte Anzahl von Arbeitsschritten auf einem einzelnen Paket erlauben. Und da sich der Umfang der IP-Messdaten für grosse Netzwerkbetreiber alleine für einen Tag im Bereich von mehreren zehn bis hunderte Gigabyte Daten bewegt, benötigen Verkehrsauswertungsanwendungen auch sehr lange für die Auswertung des Verkehrs.

Aufgrund der fehlenden Skalierbarkeit der hergebrachten zentralen Anssätze zum Messen und Auswerten von Verkehr bei steigendem Verkehrsvolumen schlägt diese Dissertation ein verteiltes Modell zur Verkehrsmessung und -auswertung vor. Sie präsentiert und evaluiert eine generische Architektur für die Verteilung von IP-Messdaten (DITA) vor, die ein Rahmenwerk für das verteilte Auswerten von Verkehrsdaten umfasst. Die Bewertung des implementierten DITA-Prototyps zeigt, dass eine Erhöhung der Anzahl im Verkehrsauswertungsprozess involvierter Knoten ein effektives Mittel zur Bewältigung eines erhöhten zu analysierenden IP-Messdatenaufkommens darstellt. Zusätzlich schlägt die Dissertation zwei Messmechanismen als konkrete Instanzen des verteilten Messvorgangs vor, die die Probleme herkömmlicher Lösungen angehen. Der erste Messmechanismus erlaubt es softwarebasierten Verkehrsüberwachungsanwendungen wie Snort [90] oder nTop [79], parallel auf mehreren Maschinen zu laufen und so die maximale Anzahl inspizierbarer Paket pro Zeiteinheit zu erhöhen. Der zweite Mechanismus ermöglicht die Identifikation des jeweiligen Benutzers und der An-

wendung, die ein IP-Paket gesendet oder empfangen hat – und nicht nur die Identifikation des benutzten Netzwerks. Dieses Werkzeug unterstützt Systeme zur Abrechnung und zum Erkennen von Angriffen in Mehrbenutzerbetriebssystemen.

Die Bewertung dieser neuartigen vorgeschlagenen Mechanismen zeigt, dass die Verteiltung von Verkehrsmessung und -auswertung machbar ist und dass sie die Grundlage einer skalierbaren Infrastruktur für IP-Verkehrsmessungen und deren Auswertung in der Zukunft sein kann.

Chapter 1

Introduction

Current high-speed links are a challenging environment for IP traffic analysis applications. Year after year during the last decade network operators observed major increase in the traffic they carry [69], [80]. Services that traditionally were delivered ontop of a dedicated network (such as telephony, or TV) now move towards an IP-based distribution. New applications allow users to create, publish, and access large amount of content in different formats: text, audio, and video. Besides the increase in number of applications, also the availability of the Internet gradually increased. Today almost all mobile devices (mobile phones, laptops) sold also have at least one network interface which allows Internet connectivity. A user browsing a webpage or watching a TV show on his mobile phone while commuting by train is a very common image these days.

Studies, such as [18] and [19], predict that the traffic increase will continue at the same pace at least during the next years. The traditionally centralized approach to traffic analysis cannot cope with the increased amount of metering data measured by large network operators, and the high packet rates experienced in the network backbone cause traffic metering devices to use sampling. The problems of centralized traffic analysis architectures arise from the fact that all metered data is sent to a central location to be stored and analyzed. Major research was done in finding sampling methods for IP packets and IP flows in order to reduce the amount of data that needs to be analyzed while keeping a high level of result accuracy. Although sampling proves to be a promising approach, there may be application scenarios foreseen, in which decisions may not be based on sampled data, e.g., usage based charging or intrusion detection systems, such as Bro [11] or Snort [90]. Moreover, traditional traffic analysis is located in the core network and cannot map the traffic observed in the network to a particular user, but rather to a particular end-node, which may have been shared by several users.

Based on these observations, this thesis investigates new paradigms to IP traffic analysis and introduces an architecture for distributed IP traffic analysis (DITA). One is the analysis of a reduced data set, by using sampling or aggregation [33]. Another possible paradigm, which this thesis investigates, is a scalable increase of computing resources that perform traffic measurement and analysis tasks. Such a distributed traffic monitoring system does allow an administrator to “upgrade” its traffic monitoring infrastructure by adding new analysis machines which take over the load of existing ones. This approach is already used for specific network monitoring applications, such as storage of flow records data. For such an application, network operators may use a separate repository for each network operation center in which they collect IP measurement data. However, such a static approach requires extra effort if data needs to be correlated between multiple repositories, so, more dynamic and flexible solutions are required.

Therefore, the main goal of this thesis is to develop an open and generic architecture for the distribution of IP metering data with the purpose of analyzing these data in parallel in a distributed system. As the evaluation of the proposed architecture and the mechanisms implemented by this thesis, which instantiates it, show, such an approach helps traffic analysis applications to handle more measurement data, and deliver their results faster.

1.1 Traffic Developments

Several studies have shown that the yearly increase of traffic observed in large network operators during the last decade ranged between 50% and 100% [69], [80], [86], with steep increases in the last years for the mobile Internet traffic segment. Responsibility for this traffic increase is shared between network operators and service providers. Network operators have not only continuously expanded their customers base, but also increased the bandwidth available to home users. On the other hand, service providers profited from the larger bandwidth users have at their disposal and have deployed new services, such as Internet radio, Internet television, on-demand video, which greatly contributed to the yearly increase of Internet traffic. A study [18] on the evolution of Internet traffic released by Cisco in June 2008 and updated in June 2009 [19] shows that this trend will continue at least until 2012 when the Internet traffic has grown approximately 75 times larger than the Internet traffic of 2002. The traffic increase not only impacts the routing and switching infrastructure of an operator, but it also affects his metering, monitoring, and accounting infrastructure which are vital operations for a modern network. Some of these operations are easy and only require to know aggregated values, such as the number of packets or bytes observed on an interface, but some are not so trivial and require more granular information, such as the number of active flows in the network, or the start time, duration, and size of every single flow in the network.

1.2 Bottlenecks in IP Traffic Monitoring

In order to properly address the challenges of high speed traffic monitoring, key problems need to be identified. Centralized traffic monitoring and analysis architectures experience bottlenecks at different stages in the monitoring pipeline (*e.g.* during metering, exporting, or analysis).

The metering process quickly becomes overloaded if the time required to process a single packet exceeds the interarrival time between two consecutive packets. In case of counting the amount of traffic (number of bytes and packets) observed on a network interface, the processing of a single packet requires updating several counters (such as a packet counter, a byte counter, unicast/multicast counters, etc.) mapped to the interface on which the packet

was observed. Although each packet might change tens of counters, such operations can be done at line speed, even in case of high packet rates, as these counters can be kept in fast memories, such as Static Random Access Memory (SRAM), or even registers or line processor caches. In case of flow accounting data about active IP flows is stored in a flow cache. Each packet triggers a lookup into the flow cache to retrieve the flow record which needs to be updated. Even with efficient flow cache management algorithms (*e.g.*, hash tables) such a process triggers multiple accesses to the main memory, where the flow cache is stored. In case of high-packet rates the time required to find a corresponding flow record entry often exceeds the packet interarrival time. As a result, not all packets can be processed [41]. For example, a router only has 40 nanoseconds to process a single packet at a rate of 25 millions of packets per second. Such packet rates can easily be achieved on a single OC768 (Optical Carrier) link which runs at 40 GB/s. Considering high-end routers operating in network backbones which aggregate multiple such OC768 links, the problem only gets worse. Typically, the memory bottleneck is addressed by sampling the packets to be inspected, which allows more time for the metering process to handle a single packet.

Another component that often experiences bottlenecks is the data exporting process from the metering point [25]. Considering the same flow accounting application, when millions of different active flows exist in the network the flow cache memory fills very quickly and flow records need to be exported in order to create space for new ones. In case of attacks, or a small flow cache memory, it is often the case that most of the observed packets create new flow records. The rate of creating new flow records can easily exceed the rate at which the exporting process can export this data, which leads to a bottleneck caused by the exporting process.

Besides metering and exporting process, problems also appear at a data collector or during analysis. Often a network operator collects metering data from multiple observation points in his network. If all data are collected by a centralized collector it may cause bottlenecks on the network link, which aggregates all data exported. In addition, if the collected records need to be stored in a persistent memory, the rate of incoming data could exceed the rate of writing in the persistent memory. An analysis application may experience a bottleneck similar to the metering process, when the rate of incoming metering records exceeds the rate at which these records are processed.

As it was observed during the recent years solving the problem of a bottleneck in the monitoring and analysis pipeline of a large network only moves the problem to another component [25]. This thesis addresses the bottleneck problem of traffic monitoring and analysis in high-speed networks in a generic way which includes mechanisms to alleviate all those bottlenecks described earlier.

1.3 Problem Statement

The main goal of the thesis is to develop an open and generic architecture which enables distributed traffic monitoring and analysis. Starting from the above observations those challenging aspects of traffic metering, monitoring, and accounting in high speed networks are investigated, and a new architectural design to handle those problems in a distributed system is proposed.

Thus, the first problem which this thesis investigates is ***what is the effect of distributed IP traffic analysis?*** As current centralized solutions to IP traffic analysis have shown their limits in processing large amounts of traffic metering data, distributing the IP traffic analysis process is one way to reduce the amount of data that needs to be processed by a single device. A generic architecture for IP traffic analysis, independent of the analysis applications running ontop of it, is important, as it is the basis for future scalable traffic analysis infrastructures. Such a system allows network operators to scale up their traffic analysis infrastructure by adding new machines, rather than upgrading or changing it completely as it is typically the case nowadays. Existing distributed traffic analysis applications such as [6], [49], [57], [67] are dedicated to specific tasks, and they are not easily adaptable to solve new problems. An ideal solution is an application-independent, highly flexible middleware, which enables development and deployment of distributed traffic analysis applications and which is able to distribute the IP metering data according to policies specified by developers of traffic analysis applications.

The second problem that this thesis investigates is ***how can the performance of existing traffic monitoring applications which run on off-the-shelve PCs be improved?*** Many traffic monitoring applications were built for the Linux operating system and make use of the *libpcap* of *libpcap-*

PFRING libraries to access the packets on the network link. It was observed that at high packet rates these libraries cause the operating system to spend most of its resources on capturing packets, while leaving less resources for the monitoring application, thus, causing an overload of the system, which eventually leads to dropped packets.

Finally, the third problem investigated is ***how to increase the granularity of IP metering data, so that an IP packet can be mapped to an individual user or even a process and application***? The traditional way to address this problem is to assume that an IP address is used by a single user at a time and have a mapping between IP address-to-user mapping all times. In case of applications, the straight-forward approach is to map ports to applications. For example, all traffic to or from 80 is web traffic. In case of users the problem arises when the end systems are multi-user capable and several users run at the same time network applications (*e.g.* background bittorrent [8] applications). In this case an IP-to-user mapping is not possible anymore, as two consecutive packets from the same IP address may be produced by applications of two different users. In case of application accounting ports are not reliable anymore in concluding the application for a given packet, as more and more applications use the Hypertext Transfer Protocol (HTTP) [45] for exchanging data (and often port 80).

1.4 Thesis Contributions

This thesis develops an open and generic architecture named DITA for a distributed system enabling the distributed storage and analysis of IP traffic. The instantiation of DITA consists of a framework for distribution of traffic metering data (named SCRIPT), and two distributed metering mechanisms (DiCAP and Linubia). The evaluation of the architecture and its prototypical instantiation shows that such an approach provides scalability, by addressing additional analysis workload with adding further nodes to perform these analysis tasks. Furthermore, a metering mechanism which instantiate the metering process specified by the architecture can assign network traffic to individual users or processes, which is in case of multi-user operating systems the only way of achieving the most fine-granular accounting data for a machine's operations. Finally, within this thesis the capturing and analysis of traffic at high packet rates using off-the-shelve, inexpensive machines is in-

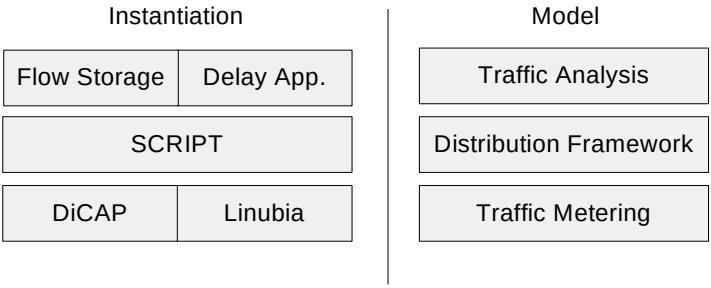


Figure 1.1: Architecture for Distributed IP Traffic Metering and Analysis

tegrated and investigated. Thus, as outlined in Figure 1.1, a proposed architecture for distributed traffic metering and analysis was designed, which integrates the distributed traffic analysis, a set of related mechanisms, and relevant interfaces. This architecture was fine-designed, implemented, instantiated, and evaluated. Figure 1.1 shows on the right side a generic model for distributed IP traffic metering and analysis, and, on the left side the mechanisms developed in this thesis which instantiate the model. Thus, as a result of this approach the following outcomes of the thesis cover:

- A generic architecture (named SCRIPT) for distribution of traffic monitoring data which enables scalable IP traffic analysis is developed. The architecture is specified based on the IP Flow Information Export (IPFIX) protocol and can be used for any traffic monitoring data which can be carried in IPFIX records. SCRIPT is both a framework for building distributed traffic analysis applications, which provides an API that can be used to build distributed traffic analysis ontop of it, and an implemented prototype platform built using the developed framework. The only condition for an application to be "deployable" with SCRIPT is to use IPFIX records as an input. Two applications have been developed ontop of SCRIPT in order to demonstrate and evaluate SCRIPT applicability and performance. The first application is storage and retrieval of NetFlow records, while the second application is for calculation of packet delays based on information transported in IPFIX/NetFlow records.
- A distributed packet capturing model, and a prototype named DiCAP were developed, which allow standard Linux PCs to capture packets on a fast network link experiencing high packet rates, by combining and

coordinating resources of multiple Linux PCs. DiCAP, introduced in Chapter 4, allows several PCs running the same monitoring application to share the monitoring workload by splitting the observed packets between themselves. Its evaluation shows that when a simple libpcap application runs on a link having a high packet rate the number of inspected packets increases from 10% (when just one PC runs the monitoring application) to 100% (if four PCs share the workload using DiCAP).

- A distributed user-based IP traffic accounting model and its prototype implementation which allows traffic accounting on a per-user, or per-process basis. The prototype shows that the approach works in network setups in which a user cannot be identified by the device that generated the traffic. The implemented prototype, named Linubia, is based on a Linux kernel module that is able to account for traffic on a per-user basis in both, IPv4 and IPv6.

1.5 Thesis Outline

The remainder of the thesis is structured as follows.

In order to grasp the current state of the art in IP traffic analysis, Chapter 2 provides the technical background and an insight into the most relevant related work in this field. It presents different approaches of representing metadata about IP traffic and discusses their advantages and disadvantages.

Chapter 3 introduces three different scenarios which motivate the need for a new concept of traffic metering and analysis. These scenarios are analyzed and used to generate a set of requirements. Based on the collected requirements a generic model for distributed traffic metering and analysis is derived.

Based on the derived model, Chapter 4 describes the SCRIPT framework for building and deploying distributed traffic analysis applications. The chapter presents the architecture of SCRIPT, including the network elements, and the protocols they use to exchange information, followed be the API which can be used to built applications ontop of it. Furthermore, Chapter 4

describes the prototypical implementation of the architecture and it describes the key interfaces between the main building blocks. As examples of distributed traffic analysis, three different applications built ontop of SCRIPT are also briefly discussed. As the evaluation of SCRIPT shows, distributed traffic analysis built ontop of the framework achieve better results compared to similar centralized applications.

As the input of traffic analysis applications is data coming from a traffic metering process, this thesis also proposes two different traffic metering mechanisms which can be used to feed metering data to SCRIPT. Chapter 5 introduces the first metering mechanism, which is a distributed packet capture system named DiCAP. The section includes the design of DiCAP as well as the most important implementation details of its prototypical implementation. Already existing Linux traffic capture and analysis applications can be parallelized using the DiCAP module in order to handle high packet rates.

A second metering system named Linubia (Linux user-based IP traffic accounting) is presented in Chapter 6. Linubia solves the problem of per-user IP traffic accounting in multi-user operating systems. The architecture of Linubia is based on the Linux kernel, but a similar approach could be used for other operating systems as well. A description of the implementation of a prototype for the Linux kernel version 2.6 is also described. Linubia offers per-user statistics for both IPv4 and IPv6 without introducing overhead in the processing of each packet.

Chapter 7 presents an evaluation of all developed mechanisms, SCRIPT, DiCAP, and Linubia. A feature analysis identifies how the functional requirements specified in Chapter 3 are provided by the newly developed mechanisms. A feasibility evaluation is also done in order to assess the impact of the newly developed mechanism on existing protocols, or metering and analysis infrastructure. Finally, the performance of all mechanisms is evaluated with respect to their overhead, limitations, and load balance.

Chapter 8 summarizes this thesis' main contributions and highlights its main results. Based on the achieved results it draws conclusions and presents an outlook to possible future work.

Chapter 2

Traffic Monitoring: Technical Background

The management of modern IP networks relies heavily on IP traffic metering and analysis. Results of such metering and analysis applications come in different forms and they are used by network operators to check for network load, detect broken links, identify illegitimate use of the network, detect network intrusions, plan for network upgrades, and many other tasks. In order to better understand how different granularities of IP metering data impact traffic analysis applications, the most common traffic representations need to be investigated. Similarly, protocols used to transport IP metering data are also discussed with respect to their advantages and disadvantages. As previous work in distributed traffic analysis shows that such an approach provides benefits in certain situations, the most relevant related work in the field of distributed traffic analysis is also introduced and discussed.

As Figure 2.1 shows, measurements taken at the network level (links, router, switches) are used to feed all the processes in the traditional FCAPS (Fault, Configuration, Accounting, Performance, and Security) [54] network management model. Data is initially collected for performance measurement and for accounting. After a processing phase in which the raw measured data is transformed into more elaborate information elements, more complex network management processes may be deployed, such as management of network performance (*e.g.*, network or service load balancing), fault management (*e.g.*, by looking into anomalies in the collected data), or security management (*e.g.*, by searching for illegal traffic in the metered data).

To provide an overview on IP traffic monitoring, the different granularity levels at which IP metering data can be collected are introduced. Advantages and disadvantages of using different traffic representations are presented. Furthermore, an investigation of two of the most used forms of traffic representations, packets and flow records, follows. Different mechanisms of data reduction, such as sampling, aggregation, and filtering are introduced, and other proposals of distributed traffic analysis are discussed.

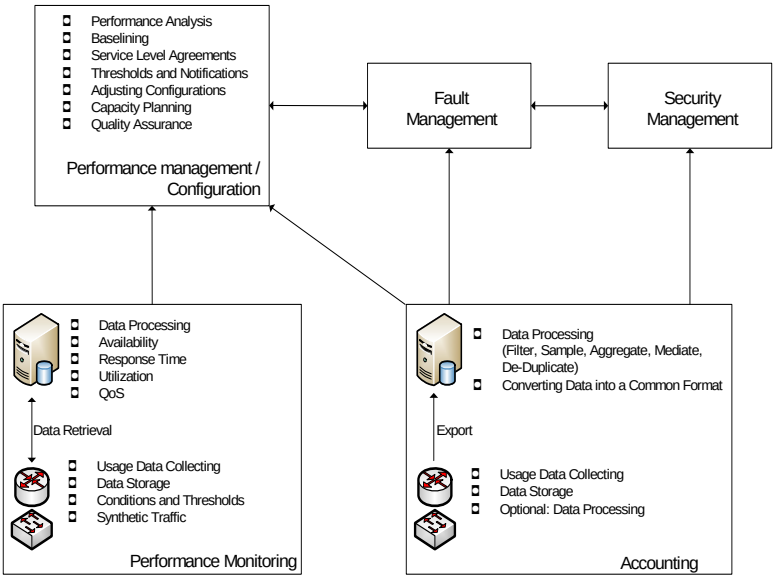


Figure 2.1: Network Management Building Blocks [24]

2.1 Data Granularity

All types of traffic analysis applications use some kind of representation of the traffic being monitored. Some applications directly use the traffic flowing through the network and inspect every single packet, other receive aggregated information (such as number of packets observed, type of traffic, statistical information), while other rely on information retrieved from routing tables, device status, and other. It can be easily observed that the type of information used for a given monitoring or analysis application dictates the amount of information used by that application and the expected detail for the results of the analysis process. For example, if packet headers are used for an analysis application, the amount of data to be processed is proportional to the number of packets flowing through the network. On one hand, a large network of an operator requires significantly more resources to run such a traffic analysis application in its core network compared to running the same application by a smaller, regional Internet Service Provider (ISP). On the other hand, some other traffic monitoring application for detecting link overload might require just the amount of traffic flowing through every link in the network every minute (such information can be retrieved using the Simple Network Management Protocol - SNMP [50]). In this case, the amount of data is independent of the amount of traffic in the network, as regardless of the traffic, every minute N readings are used, where N is the number of links in the network. This second example shows an application that has no scalability problem when network traffic increases, while the first example shows an application that is highly probable to experience scalability problems with increase of network traffic.

Table 2.1: Commonly used traffic representations

Representation	Information	Trace size (relative to the whole traffic%)
Packets	full packet	largest (100%)
Packet headers	packet header fields	large, 20 bytes per packet (10%)
Interface counters	all traffic aggregated to one or several counters	very small (constant)
Flow records	meta information about IP flows	medium 20-40 bytes per flow (1%)

Table 2.1 shows a summary of data types used for traffic representations in traffic monitoring and traffic analysis applications. It can be observed that there is a direct relation between the amount of data a particular representation produces and the granularity of information that representation has. As it can be seen in the figure, the most information about traffic is obtained by inspecting every single packet. By accessing application protocol information a large set of network attacks can be detected, such as code injections or buffer overflows, which are undetectable without access to payloads of packets. A slightly lighter traffic representation are traces of packet headers. These still keep a very granular view on the traffic, but lack the information on the data carried in IP packets. In terms of trace sizes, a packet header trace contains the same amount of elements as a full packet trace, but each element is significantly smaller, as only the IP packet header and the transport protocol header are saved. Traffic flow measurements [13] maintain about the same amount of information as packet header traces, but at a much lower cost in terms of data size. An important traffic characteristic that can only be estimated with flow records, but can be exactly computed using packet traces, is the time distribution of packets within a flow. A flow record can only return the time when the flow was active and the number of packets observed in that particular flow, but it does not specify if those packets were transferred at a constant rate or in bursts. Finally, the most compact traffic information is retrieved using interface counters. These may be used to answer questions related to traffic aggregates such as *"how many bytes were transferred in the last 5 minutes?"*, or *"is the traffic on the link increasing?"*.

As this thesis is focused on designing scalable distributed mechanisms for traffic monitoring and analysis applications, the data representations that lead to scalability problems (such as packets, packet headers, or flow records) are of main interest, while interface counters are outside the scope of this thesis. The following two sections give an overview on different aspects of packet-level and flow-level metering and analysis.

2.2 Packet-Level Metering

One straight forward approach to traffic analysis is to copy each packet observed in the network to an analysis engine which will then process it. This way an analysis application has access to a lot of information, as it vir-

tually observes every bit of data that flows through the network. Such an approach is however, a huge drawback in case of networks with high traffic. Often the time required to analyse a single packet exceeds the interarrival time of packets, so some of the packets need to be dropped and the analysis made only on a sample of the whole traffic.

2.2.1 libpcap

The library *libpcap* [61] is built for the Linux operating system and allows a process running in the userspace to "read" all the IP packets that are observed at the Network Interface Card (NIC) level.

During normal use, the Network Interface Card checks for each incoming frame the destination Medium Access Control (MAC) address and compares it to its own MAC address. If they match, the whole frame is copied in a buffer and the NIC informs the kernel that a new frame arrived and needs to be processed by the network stack. The kernel then reads the frame from the buffer, identifies the socket to which it belongs by looking to the network and transport layers and then delivers the payload to the respective socket. If the destination MAC address does not match the local MAC address, or if there is no socket to which the incoming data should be delivered, the frame is simply dropped.

When using libpcap, the NIC is set to promiscuous mode, which means that the kernel will receive each frame regardless of the destination MAC address. Once the frame is in the kernel, a copy is made and while the original frame follows the above algorithm, the copy is sent to a socket in the user-space where an analysis application can access it. The libpcap Application Programming Interface (API) allows configuration of filters based on which only the interesting traffic is delivered to the user-space application, in order to reduce the amount of data to be inspected.

libpcap-PFRING

An improved version of libpcap, called libpcap-PFRING [62], performs better at high packet rates due to better memory allocation and reduction of system calls for each captured packet. At high packet rates a high number of system calls and inefficient memory allocation makes the kernel spend more

time handling memory and context switching than handling the frames, which degrades the capture performances and leads to packet drops. The new PFRING library addresses this shortcoming by using a larger buffer for incoming frames, and by informing the kernel about new data only when this buffer is full. At the same time, it uses the *mmap()* function instead of allocating new memory for each packet.

tcpdump

Tcpdump [93] is a Linux software tool that allows packet capture from a network link and creates packet traces with different granularities (e.g. just packet headers, whole packets, or network and transport headers plus the first N bytes of payload). It is built ontop of the libpcap library and can filter traffic based on various packet attributes such as source and destination IP addresses, port numbers, IP protocol number.

```
root@n22:~ # tcpdump -c 10 -n -i eth1 port 22
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
15:05:15.937570 IP 192.41.135.198.22 > 130.60.155.30.59825: Flags [P.], seq 2009610824:2009610856, ack 4239441411, win 304, options [nop,nop,TS val 222
15:05:15.937628 IP 192.41.135.198.22 > 130.60.155.30.59825: Flags [P.], seq 32:64, ack 1, win 304, options [nop,nop,TS val 2224869852 ecr 84488816], le
15:05:15.937801 IP 192.41.135.198.22 > 130.60.155.30.59825: Flags [P.], seq 64:144, ack 1, win 304, options [nop,nop,TS val 2224869852 ecr 84488816], l
15:05:15.937859 IP 192.41.135.198.22 > 130.60.155.30.59825: Flags [P.], seq 144:192, ack 1, win 304, options [nop,nop,TS val 2224869852 ecr 84488816],
15:05:15.937915 IP 192.41.135.198.22 > 130.60.155.30.59825: Flags [P.], seq 192:240, ack 1, win 304, options [nop,nop,TS val 2224869852 ecr 84488816],
15:05:15.938165 IP 130.60.155.30.59825 > 192.41.135.198.22: Flags [.], ack 32, win 709, options [nop,nop,TS val 84488817 ecr 2224869852], length 0
15:05:15.938176 IP 192.41.135.198.22 > 130.60.155.30.59825: Flags [P.], seq 240:336, ack 1, win 304, options [nop,nop,TS val 2224869852 ecr 84488817],
15:05:15.938180 IP 130.60.155.30.59825 > 192.41.135.198.22: Flags [.], ack 64, win 709, options [nop,nop,TS val 84488817 ecr 2224869852], length 0
```

Figure 2.2: tcpdump screenshot

The data captured by tcpdump can be shown directly on screen as in Figure 2.2, or can be redirected to a file where it may be inspected later. It can be seen in the figure that tcpdump can be started with a command line and it generates human-readable output. Tcpdump does not provide any analysis of traffic, additional tools that use traffic dumps created with tcpdump are required to analyze such data.

2.2.2 Scalable Monitoring Platform for the Internet

Scalable Monitoring Platform for the Internet (SCAMPI) [28] defines an architecture for packet monitoring in Internet, which allows applications to access different types of capture devices (such as DAG cards, libpcap processes) through a unified interface called MAPI (Monitoring API). For a monitoring application SCAMPI provides an adapter abstraction which can be used to configure how packet capture takes place. MAPI, by using the available adapters starts the packet capture process and notifies monitoring appli-

cations when packets are captured. MAPI can also use adapters with built-in processing features (such as DAG cards), in these cases some of the packet processing is done on the capture cards themselves (*e.g.* flow record creation and export), and only the results of those processing tasks are forwarded to higher level monitoring applications.

2.3 Interface Counters

Interface counters are data elements which store highly aggregated metering values, such as number of packets observed on an interface, number of errors detected on a link, and so on. This data is usually kept at the metering points and is retrieved when needed. One of the most used protocol to retrieve interface counters is the Simple Network Management Protocol (SNMP).

2.3.1 SNMP

The Simple Network Management Protocol (SNMP) [50] is a User Datagram Protocol (UDP)-based network protocol used to transport metering data from network devices to a monitoring station. Typical traffic information that is retrieved using SNMP is usually highly aggregated, such as number of packets or bytes observed on an interface during a time interval, or number or packets with errors, etc. Such information can give an overview on the operation of the network with high efficiency and low overhead, but can hardly assist in many network troubleshooting activities, such as finding an intrusion or the source of a worm, which requires more granular information of the traffic.

SNMP is a highly flexible protocol which uses an extensible design based on an information model that allows definition of management information bases (MIBs). A MIB contains a collection of objects (referred by identifiers) used to manage a particular network entity (device, protocol, or application).

2.4 Flow-Level Metering

As opposed to packet-level metering, flow-level metering collects information about the on-going flows in the network. Flow-level metering may be used for a large set of traffic monitoring application scenarios, from generation of traffic matrixes [7] to intrusion detection systems [91] based on flow records analysis.

The Internet Engineering Task Force (IETF) defines a flow [83] as *a set of IP packets passing an Observation Point in the network during a certain time interval having a set of common properties.* The traditional definition of an IP flow assumes that the set of common properties is made out of:

- source and destination IP addresses
- source and destination port numbers
- protocol number

These properties are often referred to as *5-tuple*.

In the last decade flow metering became a widely deployed accounting mechanism in IP networks. Nowdays all major network hardware vendors equip their devices with flow metering capabilities. Virtually every network operator uses IP flow metering for its operation. Traffic information is collected and stored in flow records that give an overview on network usage on different levels of granularity.

During flow metering each observed packet is used to update a set of counters for the flow that it belongs to. The meta-information used to describe a flow together with the counters of that flow make a flow record. Once a flow is detected as terminated, its flow record is sent to a *collector* device in an *exporting process*. In IPFIX terminology, a *collector* is a device that hosts one or more c*ollecting processes*, while an *exporter* is a device that hosts one or more *exporting processes*. A *collecting process* receives flow records from one or more *exporting processes*.

Figure 2.3 shows the architecture of a flow metering process. The *exporter* extracts the packet header from each packet seen on the monitored interfaces. Each packet header is marked with the timestamp when the header

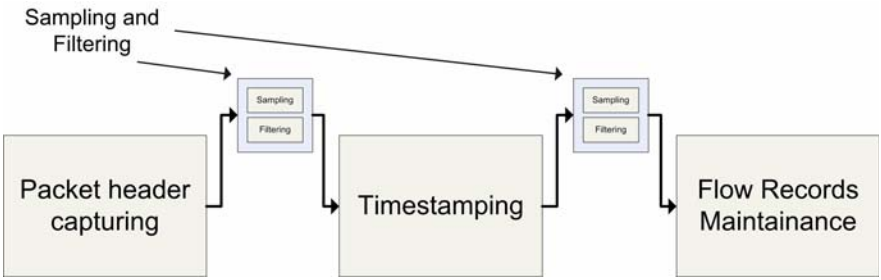


Figure 2.3: Flow metering architecture

was captured and sent to a flow management module. The flow management module has the task of creating new flow records, updating existing flow records based on the received packet headers, and exporting flow records to the flow collector. Each incoming packet header triggers an update to a flow entry. If there is no flow matching the packet header a new flow entry is created in the flow-cache memory. Once a flow record *expires* it is sent to the flow collector. A flow is considered as expired when:

- The flow is idle (no packets have been detected in the flow) for a longer time than a given threshold.
- A maximum lifetime for a flow may be defined. Once a flow reaches the maximum lifetime its corresponding flow record is exported to the collector and a new flow record is created for that flow.
- The *SYN* flag was seen in a Transport Control Protocol (TCP) flow.
- The flow-cache memory is full: a set of flow records are marked as expired and are exported to the collector. Least Recently Used (LRU) algorithms can be used to free the flow-cache memory, but heuristic algorithms are also used.

A sampling and/or filtering process may be placed at different points on the path of the packet header within the exporter.

A flow export protocol defines how flow records are transported between an exporter and a collector. The following sections give an overview on the most known flow export protocols and flow collection tools.

2.4.1 NetFlow

NetFlow [20] is an IP flow metering system and a network protocol for exporting IP flow records which was developed by Cisco Systems. It is a proprietary protocol, but it is supported by many other platforms, such as Juniper, Linux, OpenBSD, or FreeBSD. NetFlow metering processes typically run in routers and they passively collect flow information from the packets routed by a router. The NetFLow components include a NetFlow exporter (which typically is a router) and a NetFlow collector (a remote PC) which collects the NetFlow records exported by the exporter. The active flows in the network are kept by the exporter in a NetFlow cache. During the export process NetFlow records are extracted from the NetFlow cache and exported to a collector. A NetFlow record is exported in one of the following situations:

- if the corresponding flow was terminated in case of a TCP flow (by observing the SYN/ACK flags)
- if the respective flow was inactive longer than a predefined threshold
- if the NetFlow cache is full and the respective flow has an inactivity time larger than the other flows

There are different versions of NetFlow, out of which NetFlow version 5 (v5) [20] and NetFlow version 9 (v9) [22] are the most used today. NetFlow v5 uses UDP as the transport protocol and has a fix flow record format of 48 bytes, which includes IP and port numbers for the source and destination of a flow, IP protocol, TCP flags observed, the time when the first and the last packet were observed, the source and destination Autonomous System (AS) numbers, IP address of the next hop, and others.

As many traffic monitoring applications only require IP addresses, port numbers, time of the first and the last packet, and maybe just a few other fields, NetFlow v5 produces significant overhead by including fields that are not useful in every NetFlow record. NetFlow v9 addresses this problem by using templates, which define what information to include in a NetFlow record. In addition, NetFlow v9 also supports Stream Control Transport Protocol (SCTP) besides UDP and defines new parameters to be included in NetFLow records, such as Multi Protocol Label Switching (MPLS) labels, sampling algorithm, IPv6 addresses, etc.

An important difference between NetFlow v5 and v9 is the way the two versions identify a "flow". In NetFlow v5 a flow is identified by the traditional 5-tuple (IP addresses, port numbers, IP protocol). NetFlow v9 is more flexible and allows a network administrator to define the key fields which identify a flow. For example a flow could be identified only by the source and destination IP addresses and in this case would include all packets between those IP addresses regardless of the port numbers of individual packets. Besides reducing the overhead of v5 this approach has two additional benefits: a) it allows data to be aggregated at the level that a traffic analysis application requires, and 2) it allows several different formats of NetFlow records to be exported at the same time, which are useful in case of traffic analysis applications that require different type of information about traffic.

2.4.2 IPFIX

The IP Flow Information Export Protocol (IPFIX) [23] is a flow export protocol standardised by the IPFIX Working Group [52] of IETF. The standard includes a high-level architecture [87] which defines the main components and processes, and a data model [82] which describes the information model for the IPFIX protocol. The IPFIX export protocol is described in [23]. The requirements for the IPFIX architecture have been summarized in [83]. Based on these requirements several protocols have been evaluated [60] resulting in a recommendation that the IPFIX protocol should enhance NetFlow version 9 and address its shortcomings.

2.4.3 IPFIX Mediation

The problem of flow measurement system scalability as well as other problems of flow-based network metering [58] are addressed by the IPFIX Working Group in the context of IPFIX mediation. Among the problems described in [58] are flexibility of flow-based measurements, export reliability, measurement system scalability, anonymization, data retention, and others. A proposed framework for IPFIX mediation is described in [59] and introduces a high-level architecture consisting of intermediate processes (besides the exporter and collector process) which can be configure to interact and pass the IPFIX records from one to another in a pipeline.

2.4.4 Open Source Flow Metering Tools

Besides the tools embedded in routers by the major device manufacturers, a set of open source tools may be used for flow metering, exporting and collecting flow records, and analysing the collected data.

nProbe [79] is a software tool which includes a metering process (also called a probe), an exporting process, a collector process, and several embedded analysis algorithms. The metering process may be used to create flow records from the traffic observed on a Unix host. For the exporting and collection process, nProbe supports NetFlow v9 as well as IPFIX. The collector process may be used to receive flow records from one or more nProbe exporting processes, or from other exporting processes that export using NetFlow v9 or IPFIX protocols. Similarly to the probe process in nProbe, fProbe [48] is also an open-source tool for flow metering and includes an exporting process to send the resulting flow records to remote collectors. Compared to nProbe, it lacks the collection and analysis capabilities.

The cflowd [15] library was one of the first NetFlow collection and analysis tools and it was developed by the Cooperative Association for Internet Data Analysis (CAIDA) [27]. The included analysis package allowed support for capacity planing, trends analysis, characterization of workload, tracking web hosting, accounting and billing, network planning, data warehousing and mining, and security-related investigations.

Flow-tools [47] is one of the first collection of programs used to collect, send, process, and generate reports from NetFlow data. These tools can be used together on a single server or distributed to multiple servers for large deployments. The flow-tools library provides an API for development of custom applications for NetFlow export versions 1,5,6 and the 14 defined version 8 subversions. A Perl and Python interface are also included in the distribution. Flow-tools are available as open-source, but the project is no longer maintained. NfDump [77] and NfSen [78] are used to collect NetFlow version 5/7/9 records as well as IPFIX records. These tools include storage and retrieval capabilities as well as basic analysis mechanisms, such as top talkers. NfSen is used to graphically visualize network traffic from a browser window. It also includes filtering capabilities, so that traffic inspection may be tuned to specific traffic characteristics.

2.5 Data Reduction Techniques

Core routers of larger network operators today often consists of multiple ports, each capable of 10 Gbps or higher, handling aggregated traffic amounts of hundreds of Gbps. Keeping packet traces for such an amount of traffic is unrealistic because the amount of storage required would be in the order of tens of terabytes every hour. Even the number of flow records produced in such an environment reaches an order of hundreds of thousands per second. If the network traffic almost doubled every year, during the same period the memory access speeds only improved about 7-9% [81] per year. As a result, today, network operators usually reduce the traffic they inspect by using sampling. One huge disadvantage of sampling is the loss of information, which sometimes leads to poor results [10], [64]. In case of attacks, if sampling, it is easy to miss packets which caused the attack, so IDS mechanisms might not detect such attacks.

In order to reduce the amount of traffic and flow records to be inspected, three mechanisms are widely used today: sampling, filtering, and aggregation. Using data reduction mechanisms granular information about traffic is lost, but, if carefully designed and applied, they can still retain and estimate useful characteristics of the traffic, such as number of different end points, distribution of traffic, “top talkers”.

2.5.1 Sampling

Sampling techniques [2], [3], [5], [21], [26] are used to statistically infer particular characteristics of some network traffic by analysing a small part of that traffic. Statistics have shown that most of the traffic in a network is generated by a small amount of end points. Using four different traffic traces captured on public networks [42] shows that the top 10% of all flows are responsible with 85% to 93.5% of the total traffic. In addition, [65] finds that about 1% of the flows generate 80% of the total traffic. By applying statistical methods, problems like finding the largest flows or most active end nodes can be solved even in very high speed networks. In [7] the authors propose a solution to infer the IP traffic matrix from a reduced set of network metering data. Sampling methods tuned for Quality of Service (QoS) inference in end-to-end IP-based communications are proposed in [29]. In the following sec-

tions the main sampling methods and mechanisms are shortly described. Other overviews on statistical sampling for IP measurements can be found in [84] and [101].

Packet and Flow Sampling

Packet sampling is a method to select a subset of packets out of the traffic observed on a link. One of the first packet sampling methods was *systematic sampling* (or periodic sampling). In systematic sampling every *n*-th packet is sampled. The advantage of this mechanism is in its simplicity. An implementation of systematic sampling only requires a counter which is incremented by 1 at every packet. Systematic sampling could also be designed as to "sample one packet every *n* milliseconds". This way, a timer would be set to *n* every time a packet is sampled and when it reaches 0 the next observed packet will be sampled. NetFlow [22] and Juniper flow [56] implement the systematic sampling method. Due to the periodicity in sampling this method was shown to bias the results if the packets being sampled experience a periodic behaviour, such as protocol timers. In addition, as it is a predictable sampling method, it is vulnerable to attacks and manipulations. In order to address these shortcomings *random sampling* methods have been designed.

One simple random sampling method is *n-out-of-N sampling*. In this method a *population* of N packets is constructed (for example by the next N packets observed on the network). From this population a number of n packets are randomly selected. A simple way to achieve this is by calculating n different random numbers in the interval [1, N] and select the packets with the respective positions in the population set.

A category of random sampling is *probabilistic sampling*. Such algorithms assign a probability to each observed packet and then sample that packet with that probability. In *uniform probabilistic sampling* each packet is selected independently with a uniform probability *p*. This way the interval between two consecutive samples is randomized and some of the shortcomings of systematic sampling are resolved.

As opposed to uniform probabilistic sampling, *non-uniform probabilistic sampling* calculates the probability for selecting a packet based on some quantity *L*. For example, in [41] *sampled counting* is proposed, where the se-

lection probability is based on the size of the packet. Therefore, the larger the packet, the higher the probability of being sampled. If a flow *F* is considered as the sum of *n* packets with sizes L_i, *i=1..n* then the probability to sample a packet of that flow is $1 - (1-p(L_1)) * (1-p(L_2)) *...* (1-p(L_n))$. The method also states that once a packet of a flow was selected every future packet of that flow will be selected. Sampled counting is a very efficient mechanism to detect the large flows in the network. Other variations of probability calculation for non-uniform probabilistic sampling can be found in [36] and [70].

Often the amount of flow records created for traffic is so high, that a sampling process needs to select which flow records to be exported and which to be dropped. One simple method to sample flow records is to select only those records for which the respective flows have a size higher than a predefined threshold *T.* A malicious user could exploit this and split a large flow in several smaller ones. In [36] the authors propose a non-uniform probability sampling based on size of the flows. The authors investigate in the paper what is the best probability function to apply. Further investigations of the authors in size-dependant flow sampling are found in [35], [37], [38], [40].

Based on observations that flow sampling offers statistical benefits over packet sampling, but suffers from higher resource requirements, [95] introduces *dual sampling* which offers statistical performance similar to flow sampling at a computational cost similar to packet sampling, for TCP.

Trajectory Sampling

Trajectory sampling was introduced in [34] as a method to obtain information for a flow collected at multiple metering points. For example, such information could be used to retrieve the path of a flow and identify where a particular flow experiences problems in the network. Trajectory sampling assumes a high-traffic environment, where full flow accounting is not possible. It introduces the concept of sampling packets based on a hash function calculated over a packet's payload. If the same hash function is used in the entire domain then a packet will either be sampled by each observation point through which it passes, either by none. The selected packets are then sent to a collection system that afterwards correlates the data received for the same packet from different observation points and reconstruct a path for that packet. As it does not require any network state information trajectory sampling

Table 2.2: Comparison of Hash Functions [51]

Function	BOB	OAAT	TWMX	RS	Hsieh	SBOX	MD5	SHA	FNV32	NDJB	PY	SDBM	SML	BRP	JS	Ocaml	STL	AP	BKDR	Simple	DJB	CS	DEK	CRC32	MMH
Performance	++	++	++	++	++	+	-	--	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	--	++
Avalanche	++	++	+	+	++	++	++	++	+-	+-	+-	+-	-	--	+-	-	-	--	+-	++	-	+-	--	--	-

Legend:
Performance: ++ very fast, + fast, - slow, -- very slow

Avalanche: ++ all bits affected with 40%-60% probability, + all bits affected, +- all bits affected with longer inputs, - not all bits are affected, -- linear dependency

is resilient to network problems. However, the hash-based selection requires new functionality in routers, which makes its deployment difficult.

An important aspect in multi-point sample-based measurements is the synchronization of selection processes. The authors of [51] conduct a thorough evaluation of hash functions that may be used in the sampling process in order to identify which hash function is best suited for packet selection. An evaluation based on random generated packets is used to identify the hash functions which are both fast, and representative subset of the population. Their results are summarized in Table 2.2. As it can be seen, the BOB hash function achieves good results in all investigated dimensions. For a deeper understanding of those hash functions as well as the criteria used for their evaluation, the reader is referred to [51].

Sampling Accuracy

Those sampling proposals, although alleviating computational requirements of high-speed packet processing, are not very accurate in certain scenarios, where complete information is required (such as for an Intrusion Detection System - IDS, or in usage-based charging systems). The authors of [39] propose a sampling strategy that is able to select an arbitrarily small number of the best representatives of a set of flows. This technique can be used to increase the sampling accuracy when sampling is performed with limited available computational resources. Investigations have been made in detecting how sampling algorithms impact the performance of IDS: [10] and [64] show that the sampling rate directly impacts the quality of intrusion detection. The work of [36] outlines that sampling may also decrease the revenue of network operators or it may artificially increase users' bills, when sampled data is used for charging.

2.5.2 Aggregation

Aggregation, similarly to sampling aims at reducing the amount of IP metering data. In [33] aggregation is formalized as follows. Each object to be aggregated (in case of IP metering data packets, or flow records) can be represented as a set of two parts *(a, c)* where $a \in A$ is the set of fields based on which the aggregation is done, while *c* is the set of attributes that need to be aggregated. For example *a* could be the *(sourceIP, destinationIP)* pair, while *c* could be the pair *(packet count, flow size).* Such an aggregation would generate the traffic matrix between each pair of IP addresses. Starting from a set of objects (a_i, c_i) and a partition of *A* into grains $\{A_n\}$, the aggregates take the form $(An, \sum_{i,\, a_i \in A_n} c_i)$ for *n* such that $\{i: a_i \in A_n\}$ is nonempty.

2.5.3 Filtering

Filtering is the third mechanism for data reduction and is based on discarding useless data. Decision whether to drop or not a packet is made by checking one or more data values in the packet against a predefined set of allowed values. For example filtering could be applied to inspect traffic to or from a particular IP address or port number. Filtering is more efficient compared to sampling in situations in which inspection of traffic with a particular characteristic (*e.g.* web traffic) is desired.

2.6 Distributed Analysis of IP Traffic

A distributed network monitoring system called NG-MON (Next Generation MONitoring) was presented in [49]. The authors present a pipeline architecture in which traffic monitoring and analysis tasks are divided into five different phases: packet capture, flow generation, flow store, traffic analysis, and presentation of analysed data. As Figure 2.4 shows each phase may be executed by a different system (or cluster of systems). For scaling, NG-MON requires an upgrade of the pipeline element that experiences overload. The usual type of upgrade is hardware replacement, but the authors also mention the possibility of clustering a pipeline element, however without giving any further details.

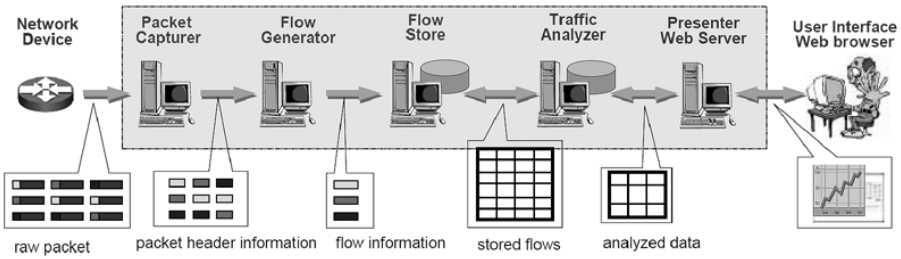


Figure 2.4: Pipeline Architecture of NG-MON [49]

Another distributed traffic monitoring architecture is proposed in [57]. The authors address the problem of flow measurement under heavy network load. Their solution includes a traffic distribution device, multiple capturing devices, and manager devices. The distribution device splits the traffic to multiple capturing devices according to predefined flow definitions specified by a user. Each capturing device performs flow accounting and reports the flow records to the managing device. The solution lacks scalability, as the distribution device is a centralized component and acts as a bottleneck, as each packet has to pass through it. Increasing the number of distributed capturing devices is not straight forward as the distribution device, the manager device, as well as the distribution rules need to be reconfigured.

In [67] the authors present a distributed architecture for monitoring web traffic. A traffic dispatcher filters traffic and mirrors all web traffic towards a set of analysis nodes. These perform analysis tasks and store results in a storage centre. The authors address the problems of fair load-balancing of analysis and efficient dispatching of traffic. Similarly to [57] this solution also suffers from a potential bottleneck, which is the dispatching node. In addition the proposed dispatching algorithm cannot maintain a fair distribution in case of denial-of-service attacks.

The authors of [6] propose IMS (Internet Motion Sensor), which is a distributed monitoring infrastructure targeted towards detecting intrusions by using traffic monitoring infrastructures in multiple networks. The authors use distribution for gathering metering information from multiple places in order to correlate information that shows a new network attack (such as a worm).

Table 2.3: Existing Traffic Metering Mechanisms

Mechanism	Granularity	Flexibility	Data size
libpcap	+++++	++++	++++
libpcap-PFRING	+++++	++++	++++
tcpdump	++++	+++	+++
SNMP	+	+++	+
NetFlow v5	++	+	+++
NetFLow v9	+++	+++	++
IPFIX	+++	+++	++
Sampled NetFlow v5	+	+	+
Sampled NetFlow v9	++	+++	+
nfSen	++	+++	+++
nTop	+++	+++	+++

DipStorage [74] and DipSIF [76] are two projects aimed to the distributed handling of NetFlow v5 records. DipStorage proposes a distributed storage of NetFlow records which also includes a distributed query language for those records, while DipSIF provides a distributed platform for sharing NetFlow v5 records between different institutions. Both proposals use central elements, are designed for dedicated tasks, and only provide limited scalability.

2.7 Discussion

Those different traffic metering mechanisms presented in this chapter are summarized and compared in Table 2.3. Libpcap and libpcap-PFRING provide the most granular traffic information. This is because these libraries pass a copy of each packet to a metering application and lets the metering application extract the relevant data. Tcpdump also offers a high granularity level, as it can be configured to extract desired fields from each packet.

SNMP counters provide a very coarse granularity because it aggregates all packets in a set of counters. NetFlow version 5 provides more detailed information compared with SNMP by allowing to collect flow-level information. NetFlow version 9 and IPFIX allow a finer grained information compared to NetFlow v5 by using templates and by allowing to customize the flow keys based on which IP metering data is aggregated. Sampled NetFlow offers the same level of granularity as NetFlow, but based only on a subset of the whole traffic. NfSen is not a metering tool by itself, but a visualization tool for NetFlow collected data. Finally, nTop provides, besides showing flow related information, also different statistics about the observed traffic.

With respect to flexibility libpcap libraries are the most flexible, as they allow applications to extract any information they need from an IP packet. Tcpdump also offers a high degree of flexibility by using configuration parameters which control the amount of information to be extracted from each packet and the interesting packets to be inspected. NetFlow v9 and IPFIX also offer higher flexibility by allowing an administrator to specify the fields to be metered and the keys based on which information will be aggregated.

SNMP is highly flexible with respect to the type of information it provides, as it allows SNMP agents to be deployed and make the expected measurements. The MIB [68] specifications are open and new MIBs for new metering information can be easily implemented. NetFlow version 5 has a fixed data format of 48 bytes which contains metering data for a flow, and can only export that information, thus is inflexible. nfSen and nTop can be configured to use NetFlow version 5 or version 9, so their flexibility depends on the underlying protocol used.

Comparing the data sizes these metering mechanisms produce it can be observed that the libpcap libraries and tcpdump by far generate the largest data sets (it is assumed that libpcap and tcpdump save full packets). NetFlow and IPFIX data sets are significantly smaller because of the aggregation of packets into flow records. SNMP produces the smallest data set out of all these metering mechanisms as the information is highly aggregated. Sampled NetFlow, NfSen, and nTop are based on NetFlow, so their storage requirements are similar. However, due to sampling, sampled NetFlow can reduce the data set it produces by modifying the sampling factor of the metering process.

The most widely used protocols for exporting metering data are shown in Table 2.4. As it was often assumed that metering data is only transported within a single network security is not present in all export protocols. SNMP only offers a basic authentication mechanism based on a community name and a clear-text password for the older versions 1 and 2c. In version 3 SNMP introduced support for multiple users and encrypted passwords. At the same time SNMPv3 also supports encryption of payload data in order to hide sensitive data from possible eavesdroppers. The NetFlow protocol assumes that both the exporter and the collector are part of the same private network and does not consider security implications. The IPFIX protocol specification relies on Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) to provide a secure transport service. Authentication relies on X.509 [55] certificates.

Table 2.4: Exporting Protocols

Export Protocol	Transport Protocol	Security	Reliable Transport	Congestion Aware
SNMP v1	UDP	no	no	no
SNMP v2c	UDP	no	no	no
SNMP v3	UDP	yes	no	no
NetFlow v5	UDP	no	no	no
NetFLow v9	SCTP/UDP	no	yes	yes (only with SCTP)
IPFIX	SCTP/UDP	yes	yes	yes (only with SCTP)

As SNMP and NetFlow v5 use UDP as a transport protocol, they cannot rely on it to provide a reliable transport service. SNMP implements a simple reliability mechanism in the application layer, but NetFlow v5 does not provide any reliability. Also because of the use of UDP, SNMP and NetFlow v5 are congestion unaware. Due to the inability to detect congestions the reliability mechanism of SNMP often aggravates congestions by re-sending data. NetFlow v9 and IPFIX favour the use of SCTP protocol for transporting data. As a result their transport service is reliable and congestion-aware.

Table 2.5: Distributed Traffic Analysis Tools

Export Protocol	Type of Traffic	Central Element	Analysis Application	Scalability
NG-MON [49]	all	yes	arbitrary	medium
IMS [6]	all	yes	IDS	low
DRTFM [57]	all	yes	Flow Statistics	low
COMS [67]	HTTP	yes	Storage/Query	low
DipStorage [74]	NetFlow v5	yes	NetFlow storage	low
DipSIF [72]	NetFlow v5	yes	NetFlow storage	low
SCRIPT [71]	IPFIX	yes	arbitrary	high

The different distributed traffic analysis tools investigated in Section 2.6 are summarized in Table 2.5. As it can be seen, all solutions use some sort of central element. For NG-MON the central element has a minimal impact on the performance, as the authors propose an optical splitter to split the traffic between multiple probes. Similarly, SCRIPT, which is the solution proposed by this thesis only uses a central element for configuration purposes. For IMS and DipSIF the central element has the role of a portal in forwarding queries to all probes. In DipStorage, DRTFM and COMS the central element plays a more important role, as it also has filtering functionality besides dispatching functionality, which impacts the performance of the system at high packet rates. With respect to the type of traffic for which those solutions were developed, NG-MON and DRTFM are used to create flow records from the traffic observed on a network link, while COMS was designed to inspect web traffic. SCRIPT works with IPFIX records. IMS can be used for different types of network intrusions, such as worms, port scans, backdoors. DipSIF and DipStorage are dedicated to storage and sharing of NetFlow records. The highest flexibility of these solutions was observed with SCRIPT which provides and API that can be used to deploy applications ontop of it. NG-MON also does not specify an analysis application, but includes a pipeline stage for traffic analysis which can include any traffic analysis application which can work with IP flow records. However the distribution in NG-MON is only for the collection of flow records, and not for the analysis application.

As the system provides a central element for accessing stored data, the performance of flow data analysis applications could suffer due to this bottleneck. In terms of scalability SCRIPT shows the highest scalability of the three solutions.

The analysis of related work identified key features for an IP traffic analysis for high speed networks. The comparison of different distributed approaches to IP traffic analysis shows that none of the existing proposals is generic and scalable enough in order to provide the basis for development and deployment of a distributed system for IP traffic analysis applications. Thus, in order to fill this gap, the following chapters propose a generic model for distributed IP traffic analysis and dedicated mechanism which instantiate this model.

Chapter 3

Models for Distributed Traffic Monitoring

In order to address those bottlenecks outlined in Section 1 in an integrated way, and to avoid the shortcomings of existing distributed approaches to IP traffic analysis describes in Section 2.6, a generic model named DITA (Distributed IP Traffic Analysis), and its attached architecture were developed in this thesis and it are introduced here. The goal of this model is to define a the characteristics and the requirements of distributed traffic analysis, as well as to outline its main building blocks. First, a comparison between centralized and distributed traffic analysis is presented which provide a high-level view on the operation of a distributed traffic analysis infrastructure compared to a centralized one. Then, based on a set of scenarios encountered in network monitoring, a set of requirements for a scalable distributed traffic monitoring

system are extracted. Following, a generic high-level model for distributed traffic monitoring is presented. The model presented here addresses bottleneck problems encountered during metering process of IP traffic as well as bottleneck problems that are specific to the data collection infrastructure or analysis applications using the metered data.

3.1 Centralized vs. Distributed Traffic Analysis

A typical deployment for a centralized traffic analysis application is depicted in Figure 3.1. A centralized collector received flow records from a set of exporters (*e.g.* routers) in the network. Upon the receipt of these records additional analysis applications use them for different purposes, such as accounting, charging, intrusion detection, network monitoring, etc. The traffic increase experienced by the network operator also translates into an increased amount of flow records that need to be handled by the collector and analysis applications. This ultimately leads to a situation in which an existing collector does not have enough resources to handle the data at the desired rate, thus the central collector needs to be replaced with a new, more powerful, but also more expensive machine. Eventually, this new machine will have similar problems in future and will have to be replaced again.

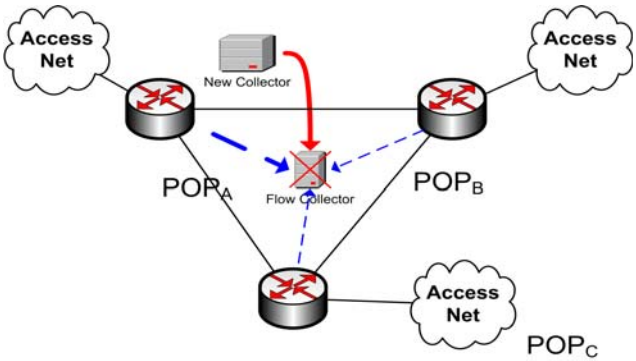


Figure 3.1: Centralized Flow Collector Replacement

A simple and direct distributed approach, such as just adding a new collector and configuring some of the routers to forward their flow records to this new collector, is not feasible, as often correlations between flow records received from multiple sources are required. Such correlations include detec-

tion of duplicated flow records in order to delete redundant data, or calculation of some network parameters based on traffic observed at multiple points. In case of two independent collectors an external component is required to perform such correlations, so ultimately the bottleneck is not eliminated, but pushed to another component.

The approach to distributed traffic analysis proposed by this thesis is summarized in Figure 3.2. Multiple data collectors form a self-organizing overlay which includes nodes that perform traffic analysis. Routers can choose any of the existing collectors to export their data to, while the overlay ensures that the exported data reaches the intended analysis applications. Using such an approach allows a network operator to address increases in traffic to be analyzed by adding new machines to the analysis overlay.

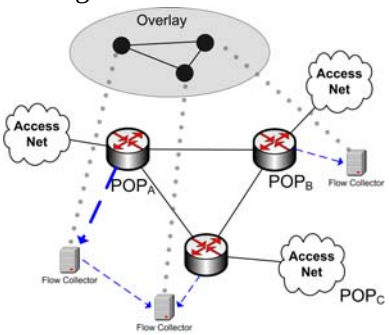


Figure 3.2: SCRIPT Approach to increased IP metering data

3.2 Scenarios for Distributed Traffic Monitoring

To extract key requirements for a distributed IP traffic monitoring seven scenarios grouped in three areas (Flow record analysis, high-speed metering, and per-user IP accounting) are selected to outline the basis of the design, as depicted in Table 3.1.

Table 3.1: Scenarios for Distributed Traffic Monitoring

Flow Record Analysis	High Speed Metering	Per-User Accounting
Data Retention	Packet Capture and Analysis on High-Speed Links	Billing for IP Traffic
Delay Measurements		User monitoring and abuse detection
Real-time Asymmetric Route Detection		Service Load Monitoring

3.2.1 Flow Record Storage and Analysis

The first set of scenarios deal with flow record processing. Three different scenarios are presented below: *Data Retention, Delay Measurement, and Real-Time Asymmetric Route Detection.* The reason for choosing several scenarios was to cover analysis applications that run off-line and require stored data, as well as applications that are real-time sensitive. The Data Retention scenario [92] is common to most network operators, as it deals with storage and retrieval of IP metering data over a longer time. Legislation was enforced in different countries [43], [30], [31] forcing network operators to keep traces of the traffic created by their users. Monitoring the delay [4] is a key element for any network operator that wants to offer high quality services which are defined by a Service Level Agreement (SLA). The third scenario - Asymmetric Route Detection (ASR) - was chosen in order to present a real-time sensitive traffic monitoring application which needs to process huge amount of data in little time.

Data Retention

Different legal regulations require that network operators keep traffic traces for some period of time. Even without such requirements network operators keep traces for a while in order to inspect them and detect possible anomalies in the traffic observed. The authors of [97] propose a mechanism to reduce the amount of data stored for data retention, but even a reduction by a factor of 20 of data that needs to be stored leaves several terabytes per month that need to be stored and processed at a later stage. A centralized approach of storing traffic traces may become a bottleneck by overloading the network link to this central repository or by sending traffic traces at a higher rate than the maximum rate at which the repository can write these traces into persistent storage. Distributing this process does distribute the network and storage load to several nodes. In addition, storing all these data at a single location means that, if the repository shows a failure, all traffic traces become unavailable and no new trace is being saved. In such a case a distributed system still enables access to all traffic traces except those ones stored on the damaged node. In case of inspecting traffic traces stored a distributed system does help by running this process in parallel on multiple nodes, making results available even faster.

Delay Measurements

A delay measurement application measures the time a particular packet spent between two observation points. The Packet Sampling (PSAMP) working group proposed a measurement delay application [103] using the IPFIX protocol for transporting metered data from each observation point. For each incoming IPFIX record the delay measurement application needs to lookup, if for the respective packet other measurements from other observation domains are in place. A high packet rate has two effects on this application: there will be more IPFIX records to be kept in main memory, which increases the lookup time, but at the same time there will be less time available to process a single IPFIX record. Distributing this application will both decrease the lookup time by storing less records in the main memory, and increase the available time to process a single record, by splitting records between multiple nodes.

Real-time Asymmetric Route Detection

Network operators often want to avoid asymmetric routes in their networks as these are usually caused by network problems such as congestion or misconfiguration. Routes are asymmetric if a flow does not traverse the same routers in one direction as in the other. To detect asymmetric routes flow records can be used by examining flow records belonging to a flow and its reverse flow, whether the same routers exported these records in one direction as in the other. To be able to do this, records belonging to a flow and its reverse flow have to arrive at a single collector from all possible exporters. Similar to those scenarios above, in case of a centralized solution the central collector has to deal with high IPFIX record rates (received from all exporters) and has less time to process a single record. Distributing this application reduces the load on a single collector and increases the time available to process a single record. It is important to note that the distribution scheme has to ensure that records of a flow and records of its reverse flow arrive at the same collector.

Shortcomings of Centralized Solutions

The major disadvantages of centralized solutions which were observed in the above investigated scenarios can be summarized as different bottlenecks due to:

- Incoming IPFIX data arrives at a rate higher than the maximum write rate of the hard disk or storage device;
- The network link bandwidth of the centralized collector is not sufficient for the aggregated IPFIX streams from all exporters; and
- In case of real-time processing, required at collector's side, processing time of an IPFIX record is higher than the records' inter-arrival time.

3.2.2 Packet Capture and Analysis on High Speed Links

This scenario stems from real problems encountered in many network monitoring research labs by researchers doing packet inspection at high packet rates [88]. Performing software-based packet inspection at high packet rates is very difficult, as the processing time of a single observed packet easily exceeds the packet inter-arrival time on that link. In such situations, sampling is used in order to reduce the number of inspected packets. Even less complex measurements, such as IP Flow accounting use sampling rates of 1/1000 on multi-gigabit links.

One of the main problems in capturing and analyzing packets on high-speed links is the very short time that can be spent handling a single packet. As shown in [98] 44% of the IP traffic observed on an Internet Service Provider (ISP) in today's Internet is made of packets with sizes between 40 and 100 byte. Assuming a 10 Gbps Ethernet link fully loaded with 64 byte packets — which are very common in voice over IP (VoIP) applications — this would translate into approximately 20 million packets per second or approximately 50 ns for handling a single packet by the packet inspection node. Capturing packets requires high-performance memory typically exceeding the speed of DRAM (Dynamic Random Access Memory) memory existing in standard PCs. Dedicated capturing cards, such as Endace's DAG cards [40], make use of the faster and more expensive SRAM (Static Random Access Memory) memory and are able to capture at those high packet rates, but they typically come at high prices.

The approach described in Section 5 of this thesis is primarily designed to achieve high capture rates by using a scalable architecture, which is able to combine the resources of multiple inexpensive off-the-shelf Linux machines. Additionally, due to its simple design, it offers a scalable traffic capture solution that could be easily embedded in cheaper hardware capture cards.

Shortcomings of Centralized Solutions

- Due to the short interarrival time between consecutive packets flowing through a high speed link software-based packet processing applications cannot process every single packet and often process only a sample of the packets observed on the link. In case of network intrusions the packets responsible for such attacks may be among the unsampled packets. A distributed system can address this problem by enabling distributed processing of packet data.

3.2.3 Per-user IP Traffic Accounting

Internet Service Providers often perform traffic accounting in order to charge their clients according to the data volume they transferred over a period of time. Such a charge does not necessarily consist of monetary units, but could be a penalty or incentive in order to reduce or increase a user's traffic (for example, some operators offer flat-rates subscriptions which include a drastic bandwidth limitation if a certain download limit in a month is exceeded). Such type of accounting process is easily doable if it is assumed that a user can be uniquely identified with an IP address at a given point in time. All that a network operator needs to do is to correlate its IP metering data with the information about which user is assigned each of those addresses. However, there are network scenarios such as multi-user operating systems in which such an assumption does not hold. On such systems multiple users might have their applications running at the same time, each generating network traffic. Using a traditional IP accounting mechanism, all that a network administrator could see is how much traffic such a system generated, but not from which applications that traffic originated and which user started each of those applications. Such a scenario is easily encountered in grid environments, where multiple users share grid resources in parallel, or in enterprise networks, where all the users of a company (or university) have access on any system in that network using a personal username and a password. The

following three scenarios motivate the need for a distributed user-based IP accounting mechanism.

Billing for IP Traffic

The first scenario in which per-user IP traffic accounting is required deals with the case of a grid infrastructure spanning across a larger network, on top of which customers run their own grid applications. A grid user will usually install its applications on multiple nodes and these run typically with the user's privileges. However, at a given time, multiple users might run their grid applications in parallel.

The grid operator may use the user-based accounting module in order to split network costs (traffic created by grid applications is typically high) among all customers based on the amount of traffic they created.

User Load Monitoring and Abuse Detection

The second scenario addresses the case of an institution, for example a university, which offers its students the possibility to use the Web for research and communication purposes, but does not want them to excessively waste precious network bandwidth for sharing videos, file sharing, and the like. The system setup is done in a way that a student can log into one of many computers at the university with his personal credentials. The user account information is stored in a centralized LDAP (Lightweight Directory Access Protocol) directory, so a specific student has to use the same user identifier (UID) in every system he logs into. A script can regularly copy usage information to a database server, where it is stored and accumulated with the traffic footprint of other users in order to detect possible anomalies in the traffic under investigation. The system administrator has the possibility to monitor network usage of students, independent of applications or the computer they use. With the help of this information he can detect and quantify abuses, suspend accounts of the respective users, or initiate further investigations.

Service Load Monitoring

The third scenario handles the identification of applications, which generate abnormal traffic. For example, on a Linux server different services may be operational, some of them may not be using well-known ports (*e.g.*, a Peer-to-Peer - P2P - bit-torrent client, which constantly changes the ports it uses to communicate with other peers). Often web servers are started on ports other than 80 which is the well known port for HTTP traffic. The reason is that such web servers are often used to deploy web services, or applications targeted to a specific group of users. On the router which connects such a system to the Internet, the administrator can monitor how much traffic was created, but can only identify applications based on port numbers. In case of applications that change these ports the use a user-based IP accounting module eases traffic monitoring for such kind of applications.

Shortcomings of Centralized Solutions

As traditional IP traffic accounting systems rely on measurement points located in network routers or switches which meter the IP traffic in the network based on the IP addresses in the IP header, they cannot map network traffic to a particular user or application. The only place where this information is accessible is in the end-node itself, which keeps a mapping between network sockets and the applications that created those sockets.

3.3 Requirements

Based on the discussion in the above examples, a set of requirements for distributed traffic monitoring and accounting are derived and summarized below:

R_1: Scalable Traffic Analysis without Sampling

The traffic analysis mechanisms defined here need to be designed in such a way that increase of traffic to be analyzed can be addressed by adding new resources to the traffic analysis process. The scalability of the solution should not come from sampling the data in such a way as to minimize the sampling error, but from distributed mechanisms that allow more data to be processed per time unit. Ultimately the traffic

metering and analysis processes should be able to cope with network links with high packet rates.

R_2: Flexibility

The distributed traffic metering and analysis mechanisms developed here shall not be designed to optimize a particular application, but shall be flexible and generic enough to allow a larger set of applications to make use of them. They shall provide the basis for developing future scalable network management applications.

R_3: Incremental Scalability

Scalability should be transparent and should not require the redeployment, or restart of the traffic analysis application. The system should be able to start with minimal resources and expand as it needs to grow, without having to go through a major upgrade in which smaller resources are replaced with more powerful ones.

R_4: High Availability

As each participant node in a distributed traffic metering and analysis application is a stand-alone machine, the failure of one node should not lead to loss of availability for the metering or analysis application in question. Fault tolerance mechanisms should be able to readapt the system to the changes introduced by unavailable nodes.

R5: Based on Commodity Components

By using commodity building components as building blocks for the mechanisms developed in this thesis, the result should be a system which can achieve better performance at a smaller price compared to a large single machine performing the same task.

R_6: Ability to detect originating end-user or processes in case of network abuse

The traffic monitoring approach presented here shall support accounting at a more granular level than the traditional traffic-per-IP approach. Depending on the granularity desired, each individual packet or flow needs to be mappable to an existing user of the system. In case of en-

terprise networks, in which a user gets network access on multiple machines with the same credentials, the accounting system needs to be able to correlate accounting data belonging to the same user and collected at different systems.

The user-based accounting model should be usable for both IPv4 and IPv6 protocols. In addition, it should also allow application-based traffic accounting.

R_7: Based on Open Standards

Any proposed solution shall use standardised protocols and standardised data formats in order to allow its integration with existing metering and analysis infrastructure and to support interoperability between with these.

3.4 Model for Distributed Traffic Metering and Analysis

To develop an integrated solution to the management problems discussed, a respective model for distributed traffic metering and analysis is designed. Figure 3.3 shows the distributed IP traffic monitoring and analysis model proposed by this thesis. It shows a layered architecture including a metering layer, a monitoring and analysis layer, and a presentation layer. The distributed metering layer includes one or more metering systems which are responsible with extracting the relevant data from the observed traffic. In order to cover also the user-based IP accounting problem this layer includes a model for general packet capture and processing, and another model for user-based IP traffic accounting.

The second layer shown in Figure 3.3 represents a distributed analysis system which enables traffic analysis applications to be deployed in a distributed environment. IP metering data is received from the metering system embedded in IPFIX records and uses internal mechanisms to forward this data to application instances that use it.

Finally, the third layer is a presentation system which allows a human administrator, or other external applications to visualize the results of the analysis process. A presentation system is dependant on the analysis application

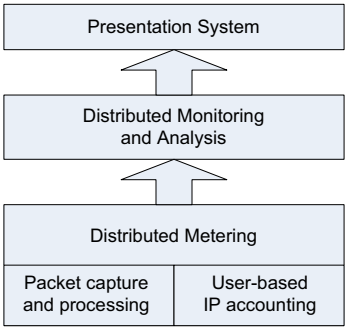


Figure 3.3: Distributed IP Traffic Monitoring and Analysis Model

and usually has a different functionality and behavior in different analysis applications, thus, it is out of the scope of this thesis.

Figure 3.4 shows the building blocks of a distributed traffic analysis system and their interrelations. In a network there are multiple *network components* (such as routers, switches, links, services, etc) that need to be monitored. The operation of these components is observed and measured by a *Meter.* One meter can measure more than a single component, for example it could measure traffic aggregated from several routers. At the same time a network component can be metered by multiple meters, for example one meter doing packet measurements, and a second doing flow measurements.

Another example of multiple meters serving a single network component is a distributed meter, having several meters, each monitoring the same network link. The metered data, once it is produced, needs to be exported to one or more *Data Collectors.* These data collectors perform limited pre-processing tasks, such as aggregation, anonymization, filtering, or encapsulation, which prepare the data to be used by traffic analysis applications, before feeding the received data to a *Traffic Processing Platform.*

The encapsulation process is of particular importance as its task is to switch the format of the received metered data (*e.g.* SNMP, NetFlow v5, Diameter, IPDR, proprietary protocols) to IPFIX which is used by the traffic processing platform. The traffic processing platform consists of one or more *Processing Units.* Each processing unit runs one or more *Traffic Analysis Applications.* It is the task of the traffic processing platform to feed each piece of metering data to the right analysis application instance.

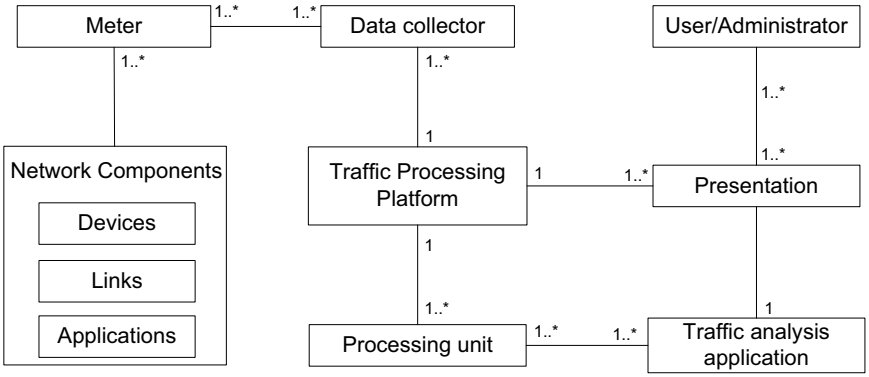


Figure 3.4: Distributed Traffic Analysis Components

The results of the traffic analysis applications are fed to a *Presentation* component which presents them to a *User* or *Administrator.* The presentation component also maintains a relation with the underlying traffic processing platform which allows it to access different traffic application instances.

The model shown in Figure 3.4 guides the design of all the mechanisms proposed in this thesis, as it models distribution in all the layers of the architecture for distributed IP traffic analysis introduced in Figure 1.1 (traffic metering, distribution framework, and traffic analysis applications).

Chapter 4

SCRIPT - A Framework for Scalable IP Traffic Analysis

Starting from the DITA model specified in Section 3.4, this chapter introduces a generic framework for distributed traffic analysis named SCRIPT [71], [76], which was developed in this thesis. SCRIPT is application independent and provides application developers an easy way to develop and deploy distributed IP traffic analysis applications. First, based on the Distributed IP Traffic Monitoring and Analysis Model presented in Figure 3.3, dedicated mechanisms for distributed IP traffic and analysis are derived. The most important design details are introduced, followed by a presentation of the SCRIPT architecture. Following, details on how three different traffic analysis application are built and deployed ontop of SCRIPT are shown.

The SCRIPT approach proposes a decentralized traffic analysis platform architecture and defines a framework for building a distribution overlay for IPFIX records for the purpose of distributed network traffic analysis. Due to the usage of the IPFIX protocol, SCRIPT can be used to distribute any kind of metered IP data as long as it can be transported in IPFIX payloads, since the operation of SCRIPT is not bound to a particular IPFIX template. By doing that, SCRIPT can replace centralized architectures for IP traffic monitoring and analysis which require and use granular information about the traffic (such as packet headers, or flow records) and which do not scale well and use sampling when deployed in high-speed network environments.

In order to avoid confusions, the following terms are defined:

Definition 4.1: A ***SCRIPT network*** is a collection of nodes, built using the SCRIPT framework for the purpose of deploying one or more distributed traffic monitoring and analysis applications.

Definition 4.2: ***A SCRIPT platform*** is an instantiation of a SCRIPT network.

Definition 4.3: A ***SCRIPT node*** is a node participating in a ***SCRIPT network***.

Definition 4.4: A ***SCRIPT application*** is a distributed IP monitoring and analysis application built using the SCRIPT framework and deployed within a SCRIPT network.

Definition 4.5: The ***SCRIPT framework*** is a collection of libraries and abstract classes definitions which allows for a SCRIPT platform to be deployed and SCRIPT application instantiated ontop of a SCRIPT platform.

The following sections describe SCRIPT in detail. First, a SCRIPT distributed traffic analysis model is derived from the generic model presented in Section 3.4. Following, the most important design and implementation details are given. Once the internal operation of SCRIPT is uncovered the SCRIPT API exposed to traffic analysis applications is presented. In order to better visualize the advantages of SCRIPT the scenarios introduced in Section 3.2.1 are revised and mapped onto the developed architecture.

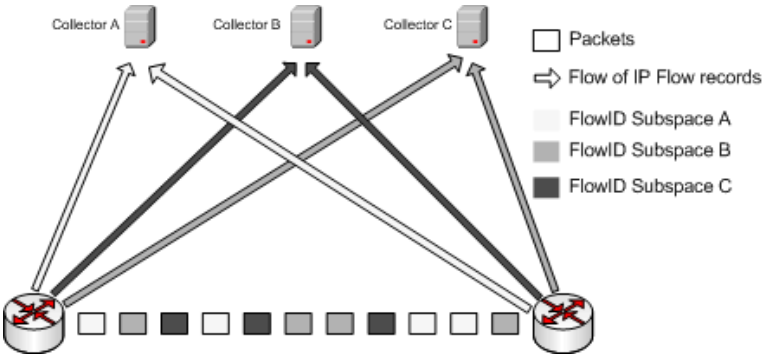


Figure 4.1: Distribution of Flow Records

4.1 SCRIPT Model

Following the distributed model introduced in Section 3.4, the key approach for SCRIPT is determined by the distribution of IPFIX data processing to multiple nodes. Each IPFIX record is assigned an identifier by applying a hash function to several fields of the record, *e.g.*, IP addresses, port numbers, or the IP protocol. Based on its identifier, each IPFIX record is sent to one of the multiple analysis nodes to process it. Figure 4.1 shows an example of the proposed mechanism for the distribution of IP flow records to multiple data collectors. The figure shows three data collectors which receive flow records from two metering points (in this case two routers) in the network. Each collector is responsible with flow records that have their identifier in a particular range (depicted in figure as light gray, gray, and dark gray). This simplified example assumes that each metering point knows in advance the way the identifier space is divided between the three collectors, so that each exporting process sends each flow record to the rightful collector. SCRIPT does not make this assumption and provides a communication mechanism between the different collectors that allow an exporting process to send its flow records to any of the three collectors and leave the collectors to forward a flow record to the right destination.

Figure 4.2 shows a high-level representation of a distributed system for IP traffic monitoring and analysis. A set of network meters (such as routers) are responsible with traffic metering according to policies configured by a net-

work administrator. This metering information is then forwarded to one of the nodes participating in the distributed traffic monitoring and analysis application. Each participant node is at the same time a data collector and a processing unit. At the same time each node runs one or more traffic analysis applications. Depending on the specifics of each traffic analysis application, each node also contains a presentation system and allows a user to access the analysis results or configure the analysis application. In order to avoid bottlenecks and problems observed in previous work, such as [57] and [67], exporting devices should be able to use any participating node for sending its metered data. In the example shown in Figure 4.2 router R_1 chose node N_1 to send its data, router R_4 chose node N_3 to send its data, while routers R_2 and R_3 both chose node N_2 to send their metered data. SCRIPT framework does not impose any restriction on the type of metered data, but only on the protocol which is used to export this data. It is assumed that either the IPFIX, or the NetFlow exporting protocols are used. Within a SCRIPT network all metered data is exchanged between SCRIPT nodes using the IPFIX protocol.

Each SCRIPT node has an internal organization as depicted in Figure 4.3. A collecting process is responsible with receiving IPFIX data from exporters or other SCRIPT nodes. An exporting process is responsible with forwarding

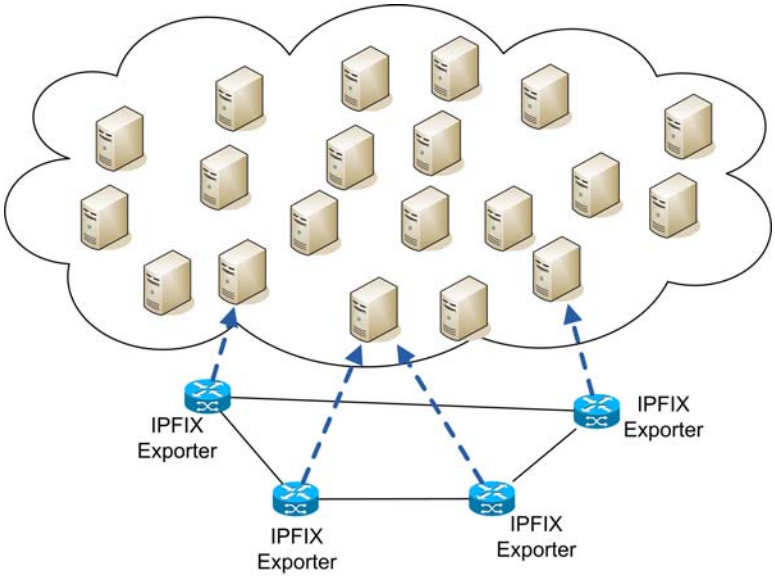


Figure 4.2: Distributed Traffic Analysis

IPFIX data to other SCRIPT nodes. The SCRIPT middleware contains all the mechanisms for forwarding an IPFIX record within the SCRIPT network, while an API may be used by application developers to build SCRIPT applications.

Within a SCRIPT network each node runs the same set of SCRIPT applications, and one of the main goals of the distribution process is to forward each piece of traffic metering data to the application instance that requires that data.

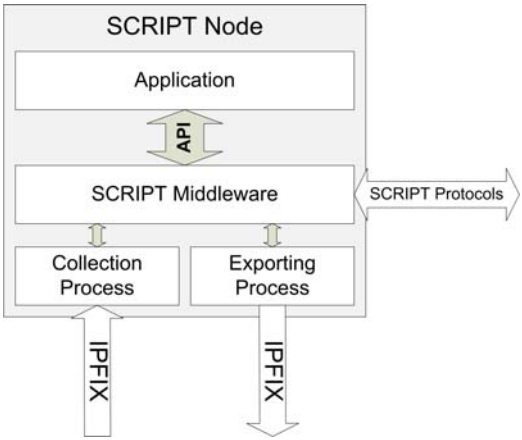


Figure 4.3: Node architecture

4.2 Design Dimensions

Based on the requirements specified in Section 3.3, a set of design dimensions specific to SCRIPT are described in the following subsections. These include the most important functionality that need to be addressed by the design of a generic framework for distributed traffic analysis.

4.2.1 Self-organization

One of the main goals of SCRIPT is to address changes in network traffic volume by simply adding or removing computing power. However, this operation should be simple and fast, and should require minimal human inter-

vention. When a new computing node is added to an already existing SCRIPT network it needs to be able to retrieve the parameters required to operate from the network itself. In case of a node leaving the SCRIPT network, a mechanism should be in place that detects such an event and takes the necessary measures in order to continue the proper functionality of the SCRIPT network. Thus, auto configuration needs to be a key feature of its design.

4.2.2 Forwarding of IPFIX Records

In case of centralized IP monitoring and analysis applications all IP metering data flows towards a single application instance which then has access to it and is able to correlate different pieces of information in order to get to produce the desired result. In a distributed system, IP metering data will be distributed to multiple nodes, and access to information is not as easy as in a centralized system as different IP monitoring and analysis application have different requirements with respect to how the input data is used. For example, an application that measures delay experienced by a packet between two routers, based on IPFIX information received from those routers, needs to have access to both IPFIX records belonging to the same packet in order to calculate the delay, which requires that all IPFIX records having similar 5-tuple are forwarded to the same node. A different application that maintains statistics about the most active downloaders may require that all IPFIX records having the same destination IP address to be forwarded to the same node. Therefore, forwarding of IPFIX records between nodes is not trivial and should always be related to the application or applications that require that record. Details about the forwarding mechanism of SCRIPT are given in Section 4.4.5.

4.2.3 Support for Multiple Applications

Often network operators use several monitoring and analysis applications at the same time, and some of those applications might share the same IP metering data. SCRIPT design allows the deployment of multiple applications in the same distributed system. As different applications might require different data sets (for example they might require IPFIX records of different IPFIX templates) SCRIPT uses a mechanism that allows each application to specify the type of data (one or more IPFIX template IDs) it requires. Simi-

larly, an application can also define routing rules for the IPFIX records it uses.

4.2.4 Template Correlation

When IPFIX records originate from multiple sources it is important to be able to correlate similar data. An IPFIX template ID is unique only for an exporting session, which means that is often the case that the same IPFIX template definition configured on two different exporters shall produce IPFIX records with different template IDs. Moreover, on the same exporting device, the same template might produce a different template ID after a reboot. SCRIPT introduces a template correlation and coordination mechanism that assigns each IPFIX record a template ID which is globally the same for each template definition, regardless of the template ID which was used by the exporter. The template correlation and coordination mechanism is described in detail in Section 4.4.

4.2.5 Network Problems

As SCRIPT is designed to operate over an IP network, its participants can be geographically distributed, so communication between nodes may experience the inherent problems of IP networks, such as congestion, unavailability, etc. Besides these problems, nodes participating in a distributed system could be required to temporarily stop their activity, for example due to a problem, or due to an operating system or hardware upgrade. Such temporary problems need to be identified and properly handled. SCRIPT implements a *temporary flow record handling* mechanism that allows for temporary storage of IPFIX records inside the SCRIPT network in case of a sudden disappearance of a node.

4.2.6 Export Protocol Compatibility

Although IETF standardized the IPFIX protocol for exporting flow records, Cisco's NetFlow v5 and v9 are still the most widely used export protocols by network operators. In order not to restrict SCRIPT to monitoring and analysis applications that use IPFIX, SCRIPT implements support for IPFIX, NetFlow v5, and NetFlow v9.

4.3 Framework Design

The SCRIPT framework design described here consists of the architecture of the SCRIPT network, a Central Configuration Repository, the SCRIPT Node architecture, and the mechanisms behind routing of IPFIX records. These major components, their interactions, and other key design decisions are detailed in the next sections.

4.3.1 Assumptions

The assumptions stated and justified below define the set of key conditions which will be taken as granted and on which the proposed approach will be based on:

- Full flow accounting in high-speed networks is too expensive in terms of cost and resource usage to be done in the router. Although sampling mechanisms can be applied, there will always be a trade-off between traffic analysis performance and accuracy.
- Using a small fraction of the total bandwidth of an Internet Service Provider's (ISP) network for traffic analysis operations is feasible. Clearly, there is a trade-off between the use of local processing capacities and the use of bandwidth for the transmission of measurement traffic. The actual traffic overhead will depend on the amount of processing power locally available, *e.g.*, within a particular point-of-presence (POP) as well as the use of sampling methods.
- Nodes might be unreliable, but do not act malicious. Nodes may join and leave the analysis network at any time. This will cause reliability and robustness issues that have to be solved.

4.3.2 Network Architecture

A SCRIPT network is organized as a Kademlia-based P2P overlay [66], as shown in Figure 4.4. Routers (R_1 - R_5) meter network traffic and export the metering results as IPFIX records. In the following IPFIX is assumed as the exporting protocol, but, SCRIPT also implements support for NetFlow v9 and NetFlow v5 protocols, so it can be used also with exporters that do not yet fully support IPFIX. SCRIPT nodes (N_1 - N_8) build a P2P overlay

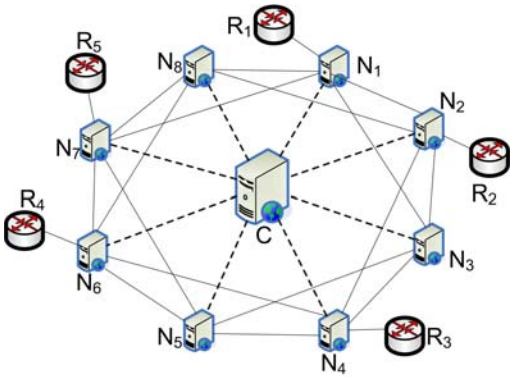


Figure 4.4: Network Architecture

network, they receive IPFIX records from routers, and they distribute these records in the overlay. Traffic analysis applications (*e.g.*, delay measurement or flow record storage) are running on each SCRIPT node, so the SCRIPT network can also be seen as a collection of SCRIPT application instances. If the SCRIPT network consists of *n* SCRIPT nodes and if there are *k* SCRIPT applications deployed, then the number of application instances running in a SCRIPT network is $N_{app} = n*k$. The goal of the SCRIPT network is to deliver each IPFIX record to the application instance that needs it. For example, if two SCRIPT applications A_1 and A_2 use two different IPFIX templates T_1 and T_2 respectively, then no IPFIX record having T_2 as template should be forwarded to an instance of A_1.

Often traffic monitoring and analysis applications require correlations between data sets. For example, a delay measurement application needs to correlate data metered at two measurement points in the network in order to find the two observation timestamps for a given packet based on which the delay is then calculated. The difference between a centralized delay measurement application and a distributed delay measurement application is that while in case of a centralized system there is only one application instance that has access to the whole volume of data, in case of a distributed system, these data is no longer available at a single location, and the two metering records required to calculate the delay of a given packet reside at two different locations. SCRIPT addresses this problem by implementing a flexible routing mechanism which always tries to “push” an IPFIX record to the node that actually needs it. In the case of the delay measurement applications, both me-

tering records for a given packet are routed by the SCRIPT network to the same application instance.

Each SCRIPT node has two main tasks: (1) to forward incoming IPFIX records to other nodes, so that each record is delivered to the application instance that needs it, and (2) to deliver a subset of the incoming IPFIX records to one or more monitoring and analysis applications running locally on that node. As Figure 4.4 shows, each router can choose any SCRIPT node of the P2P overlay to forward its IPFIX records. Starting at that SCRIPT node, a flow record routing process starts that assures that the intended application instance to process an IPFIX record will receive that record.

A SCRIPT network includes a Central Configuration Repository (CCR), which is involved in the bootstrap process (c.f. Section 4.4.7) as well as in the management of different configuration aspects of the SCRIPT nodes, but it is not involved in forwarding of IPFIX records.

4.3.3 Overview of the P2P Routing Process

As already mentioned in Section 4.3.2, a SCRIPT network is organized as a P2P overlay. Each node has an identity and knows about the identities of some other nodes, which are used during the routing process. In order to decide which node should process a given IPFIX record, a hash code is generated for that record. Based on the generated hash code, the IPFIX record is then forwarded according to the Kademlia routing protocol until the record arrives at the node running the SCRIPT application instance responsible with the processing of that record. An overview on the Kademlia routing protocol is given below, while the detailed description by its authors is found in [66].

Each node has a k-bit identity and maintains a set of k buckets. Each bucket holds details about other nodes as follows: the i-th bucket contains nodes which have the first i bits of their identity equal to the identity of the node having that bucket. Whenever an object needs to be routed in the P2P network (an IPFIX record in case of SCRIPT) a hash code is calculated for the object. The purpose of the P2P routing process is to deliver (store) the object to a set of m nodes having the closest ID to the object's calculated hash value. The value of m specifies the redundancy factor and allows content to remain in the P2P network even in case of nodes leaving. the network.

As opposed to traditional P2P networks targeted towards end-users that may join-and-leave often, SCRIPT is targeted towards long-running applications in reliable environments. A usual use-case for SCRIPT is a network operator instantiating a SCRIPT network using computational resources in several different locations. Therefore, an assumption is made, that churn rate (the rate of nodes joining and leaving) is very low. It is assumed that nodes are added to the overlay when the traffic increase requires additional computation power, while node removals (due to hardware, software, or network problems) are rare. Having this assumption, SCRIPT uses a redundancy factor of 1, which leaves applications to decide how to implement redundancy if they require.

The way the hash code is calculated, combined with the way node identities are generated dictate how many IPFIX records a particular node receives. Figure 4.6 shows an example of a simplified SCRIPT network having four nodes (black circles), each of them having an 8-bit identity. In this network a set of IPFIX records (gray rectangles) are distributed. According to the routing mechanism *node_17* will receive all records for which a hash value between 1 and 17 was calculated. Similarly, the other nodes *node_33*, *node_58*, and *node_0* will receive records with the hash value in the ranges [18, 33], [19, 58], and [59, 0] respectively. It is shown in the figure, that if the hash values are uniformly distributed, the larger the distance to the next neighbor, the more records have to be processed. The example also shows a larger set of records for which their calculated hash values collide within a small interval. Such situations appear due to a denial of service, or port-scanning and are often the result of a bad hash function.

The first problem is directly addressed by SCRIPT by assigning node identities in a way that tries to keep their distribution as close as possible to a uniform distribution (cf. Section 4.4.7). The second problem needs to be addressed separately by each SCRIPT application, as the application defines the hash function to be applied for an IPFIX record (cf. Section 4.4.5).

4.3.4 SCRIPT Node Architecture

The SCRIPT node represents the key component of the SCRIPT architecture (cf. Figure 4.4) and represents a computing device participating in the SCRIPT network. In Figure 4.4 nodes N_1-N_8 are SCRIPT nodes.

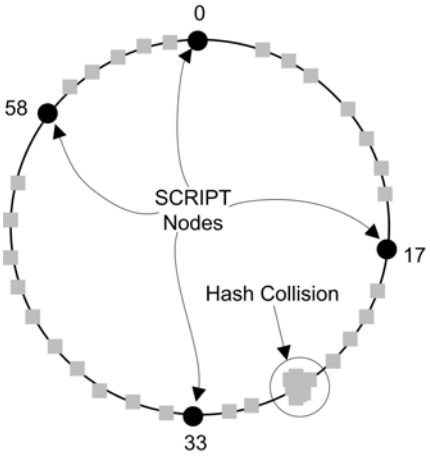


Figure 4.6: P2P Identities

SCRIPT Node Identity

Each SCRIPT node has a 64-bit identity, as shown in Figure 4.5, which is assigned by the *CCR* during the bootstrap process (cf. Section 4.4.7). An identity consists of a 32 bit unique node identifier, 16 unused zero bits, and a 16 bit area ID. The *CCR* has the task of assigning the unique node identifier.

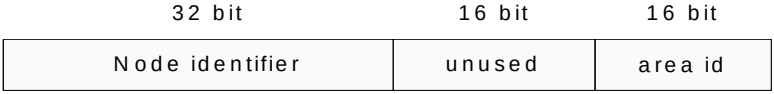


Figure 4.5: SCRIPT Node Identity

The area ID of a node identity is used to organize nodes according to geographic location in order to optimize the overlay routing, or to build logical overlays. The usage of *area id* is optional and is intended for extensions of the prototype.

IPFIX Record Routing

Forwarding of IPFIX records in SCRIPT is done using a routing function. Analysis applications can have different requirements with respect to how IPFIX records are routed. For example, a delay measurement application requires that all data exported for the same packet at different observation

points is forwarded to the same node, while a traffic matrix calculation application requires that all records corresponding to the same *(source, destination)* pair are forwarded to the same node.

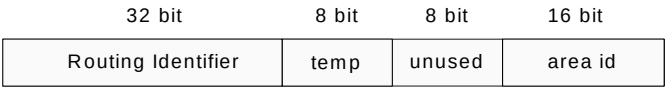


Figure 4.7: Routing Hash ID

Therefore, the routing function is a hash function applied to some of those fields of a flow record: *Hash(f(record fields))*, where *f()* is a function that enables operations on the record fields before calculating the hash value. For example *f()* can be a logical AND operation on the source and destination address. The result of the routing function applied is a 32-bit identifier based on which the node, responsible for processing of that record, can be found. Based on this 32 bit identifier, the next hop of the IPFIX record is calculated using the Kademlia protocol [66]. If a next hop cannot be found, the IPFIX record is processed locally. The routing identifier is included in every flow record in a 64 bit field called routing hash ID as shown in Figure 4.7. Besides the routing identifier, the routing hash ID field contains 8 bits that are used to create temporary routing hash IDs (cf. Section 4.4.5). The next 8 bits are unused, while the last 16 bits are set to an area identifier, which will cause the flow record of being routed only to SCRIPT nodes in that area (for example due to privacy issues).

The flow record routing process is described in Figure 4.8. Upon the receipt of a flow record a SCRIPT node retrieves the template ID for that flow record. Based on the template ID the node knows if the flow record was received from another SCRIPT node, or from a non-SCRIPT exporter (such as a router). If the flow record was received from another SCRIPT node, then the *Routing Hash ID* is already present as a field in the record and can be retrieved from there. Once the *Routing Hash ID* is available, the flow record can be forwarded to a routing process.

If this is the first SCRIPT node to process the flow record then first, three new fields are allocated for storing the *Routing Hash ID*, the *Exporter ID*, and a *Time* value which represents the time when the record was exported by the original exporter. If the reason for including the *Routing Hash ID* is clear, the other two fields are required in order not to loose that information. IP-

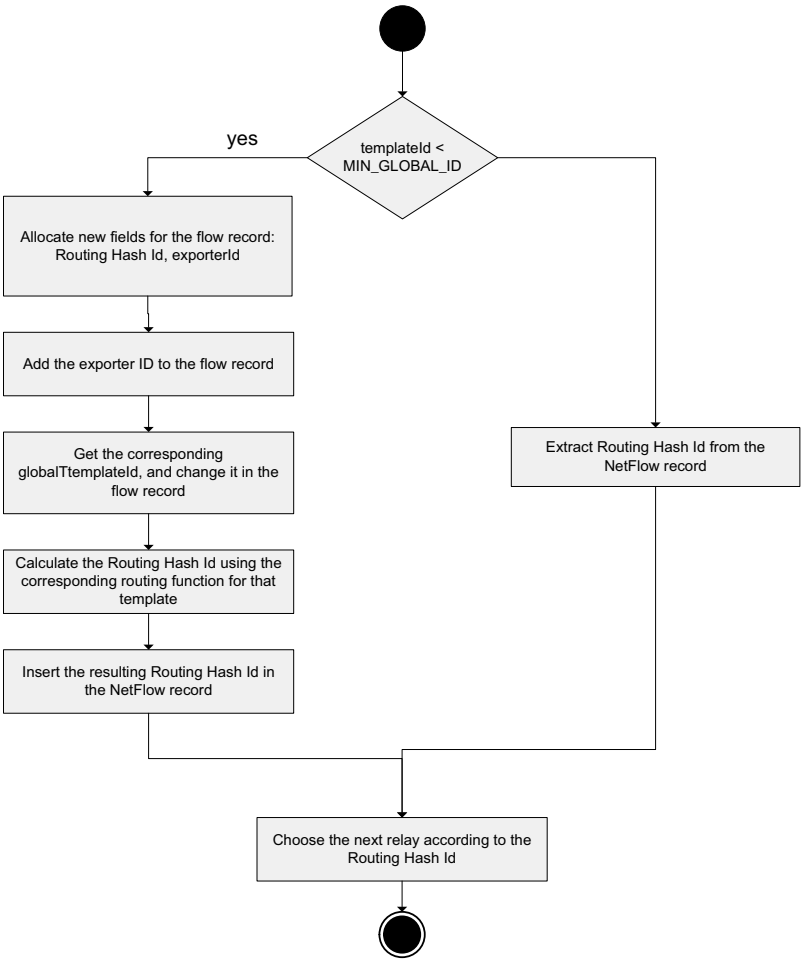


Figure 4.8: Flow Record Routing

FIX, as well as NetFlow, carries the exporter ID information and export time in the IPFIX (NetFlow respectively) packet header. This information is common to all flow records in a given packet. However, in SCRIPT is often the case that two flow records which were exported by a non-SCRIPT exporter in the same packet will be forwarded separately in the SCRIPT network, which means that the information which was originally common to all flow records needs to be copied to each individual flow record. Once the new fields are allocated, the *ExporterID* and *Time* information is copied in their

respective fields. Next, the SCRIPT node identifies the SCRIPT template corresponding to the template used by the non-SCRIPT exporter and changes the template ID in the flow record. Each SCRIPT template has a corresponding routing function which specifies how to calculate the *Routing Hash ID* for a flow record of that template. Based on the identified SCRIPT template the corresponding routing function is called and a routing identifier is generated and placed in the respective field of the flow record. Next, the flow record can be delivered to a routing process which decides what to do with it, whether to send it to another SCRIPT node, or process it locally.

Support for Analysis Applications

In order to deploy an analysis application in SCRIPT, an application ID (App_{ID}) is chosen for it and the template (or templates) for the IPFIX records that will feed this application need to be known in advance. Respective templates are configured in the *CCR* and are mapped to the App_{ID} chosen. In addition, for each newly defined template, a routing function needs to be specified. Whenever an IPFIX record is received by a node, the routing function specified for the respective template is used. If the record has to be processed locally, based on the template of the record, the App_{ID} for those applications that use that template are retrieved and a copy of the record is delivered to each of those analysis applications.

Node Architecture and Functionality

The SCRIPT node architecture (cf. Figure 4.9) consists of three main blocks: Management, Routing, and Flow Processing.

The *Management* block consists of a *Control Messaging* component, which handles all communications of a node, a *P2P Management* component, which handles joining and leaving of nodes, and a *Controller Unit*, which orchestrates the operation of all components of a SCRIPT node. In addition, it defines an Application Programming Interface (API), which allows applications to be built on top of SCRIPT.

The *Routing* block includes an *IPFIX Collector*, which handles the receipt of incoming IPFIX records, a *Flow Records Router* that is responsible for

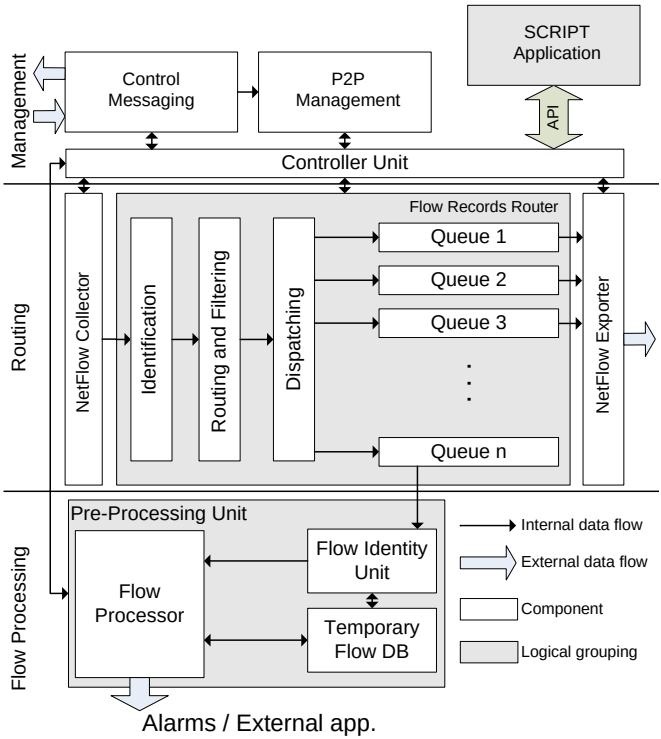


Figure 4.9: SCRIPT Node Architecture

finding the next hop of an IPFIX record, and an *IPFIX Exporter* component that is used to send IPFIX records to other nodes.

Once an IPFIX packet is received by the *IPFIX Collector* component, respective IPFIX records are decapsulated and passed to the *Identification* component. For each record, the *Identification* component checks, if the template ID represents a GTID (Global Template ID). If so, the record is passed directly to the *Routing and Filtering* component. If the template ID is not a GTID the *Identification* component checks, if a mapping of (template ID, exporter) pair to a GTID exists. If there is no such mapping, a request for such a mapping to the *CCR* is made using the *Control Messaging* component. If such a mapping does not exist on the *CCR* either, the IPFIX record is dropped as well as all future records having that template ID, until an IPFIX record with the template definition for that template ID is received. When such a new template definition is received, it is forwarded to the *CCR* which

returns a new GTID to be used for it and a routing function to be used with that template. Additionally, the *CCR* stores the new template ID and GTID in its mapping table. The final task of the *Identification* component, in case of IPFIX records with template IDs set by exporters, is to change these IDs with the corresponding GTID and set an internal flag (F_{TC}) for that record, specifying that this change was just performed locally.

Once an IPFIX record arrives at the *Routing and Filtering* component, the F_{TC} flag is checked. If it is set, a new 64 bit field is added to the IPFIX record, representing a routing identifier (R_{ID}) and containing a value calculated by applying the corresponding routing function to that IPFIX record. This identifier will be used by all further SCRIPT nodes to route the IPFIX record. If F_{TC} is not set, the R_{ID} is not calculated, but read from the record.

Based on R_{ID} and the P2P routing information, the next hop node is selected. If no next hop is found, this record is delivered to the local *Flow Processing* block. If a better candidate than the local node is found, the IPFIX record is marked to be delivered to that node and is put in the outgoing *Queue* by the *Dispatching* component. The *IPFIX Exporter* periodically reads all *Queues* and sends records to the next hop nodes.

The *Flow Processing* block includes a *Pre-Processing Unit (PPU)*, which dispatches each record that has to be locally processed to analysis applications expecting that record. When an IPFIX record arrives, the *Flow Identity Unit (FIU)* identifies these applications, which require the respective record, based on the template ID of the record, and the FIU passes the record to the *Flow Processor*, which notifies those applications by sending a copy of the new record. The *Temporary Flow DB* is a special application (cf. Section below). In addition to delivering flow records to application, the Pre-Processing Unit implements additional tasks (such as data aggregation, or sampling) that can be applied before records are delivered to the SCRIPT application.

Finally, an external SCRIPT application receives flow records from the *Controller Unit* via the SCRIPT API.

Temporary Handling of IPFIX Records

Due to a number of reasons (such as loss of connectivity, overload, or congestion) a SCRIPT node may become unavailable for some time. Such

situations are detected by nodes connected to the node experiencing problems. In these cases, IPFIX records that have to be routed to such a node, are temporarily stored by other nodes, which try after some time to deliver them.

In order not to overload a single node with all the extra workload corresponding to the missing node, the respective flow records are distributed to up to 256 other SCRIPT nodes as follows. The distribution is done by changing the *Routing Hash ID* so that those flow records are routed to other nodes. The detailed mechanism works as follows: when a SCRIPT node has to forward a record to another SCRIPT node which is known as being temporarily unavailable it changes the first 8 bits of the *routing hash ID* and the record is re-routed using the new *routing hash ID*. The first 8 bits are changed by generating an 8 bit number and then performing a XOR operation between the original first 8 bits of the *routing hash ID* with this number. The resulting value replaces the first 8 bits of the *routing hash ID* and the generated number is placed in the 8 bit *temp* field of the *routing hash ID*. Every SCRIPT node can infer that *routing hash ID* of a flow record is temporary by checking the *temp* field of the *routing hash ID*. If this field is not zero, then the flow record needs to be stored temporarily.

Due to the change of routing information described above, the node responsible for processing that record is changed. The format of the routing hash ID contains all information required to identify, if a record has the original routing hash ID or a temporary one. It also contains the information needed to reconstruct the original routing hash ID when required. As soon as the IPFIX record arrives at the new SCRIPT node responsible with its processing, by showing the temporary routing hash ID the record is placed in a *Temporary Flow DB* on that node. After a time interval, the node reconstructs the original routing hash ID and re-injects the flow record in the routing process which will deliver the flow record to the originally responsible node. Once a temporarily stored record is re-injected in the routing process, it is deleted from the temporary storage. If problems on that node are not solved and the above process is repeated until either i) the node becomes available, or ii) the node is marked as permanently unavailable.

4.3.5 Central Configuration Repository

Besides SCRIPT nodes, the SCRIPT architecture also contain a Central Configuration Repository (*CCR)*. The CCR is responsible with management of SCRIPT nodes, supporting the bootstrap process, and with managing of flow templates and SCRIPT applications.

Peer Awareness

The *CCR* also monitors all nodes and periodically checks, if nodes are "alive". Whenever a node is detected as being unavailable, the *CCR* informs all other nodes about the change. Thus, the unavailable node will be removed from the overlay and no flow records will be sent to it any more. An additional functionality of the *CCR* is the distribution of application-specific messages to applications running on specific nodes, or to all application instances on all nodes. For example, such messages are queries sent by a network administrator to the flow storage application. The query is received by the *CCR* and sent to all participating nodes subsequently.

Node Bootstrap

The bootstrapping process takes place during the joining phase of a new SCRIPT node. An addition of a new node triggers a change in the topology of the overlay network. During bootstrapping a node is assigned a position in the overlay and follows a learning phase in which it collects information about other participants in the respective SCRIPT network.

During the identity assignment, assigns a node ID following a uniform distribution. This will result in a nearly equal number of flow records received by each node. For example, for the scenario depicted in Figure 4.4 these eight nodes will see node identifiers starting with the following byte 1F, 3F, 5F, 7F, 9F, BF, DF, and FF.

Template Management

The *CCR* stores flow templates and their mapping to analysis applications. One problem identified when dealing with IPFIX records exported by different exporters was that the same template definition received different template identifiers on those exporters. In order to address this problem,

SCRIPT uses the concept of a Global Template ID (GTID). Each SCRIPT node maintains a mapping between the pair (exporter ID, template ID) and GTID. At the entry point in the SCRIPT network, the template ID is changed to GTID for each IPFIX record. Thus, two IPFIX records having the same template definition and exported by different exporters will always have the same GTID, although the template IDs that these exporters used may have been different. Each node can detect, whether the value in a template ID field is a GTID by looking at the first bit of that value. If the first bit is "1", the value represents a GTID, otherwise it is a template ID set by an exporter, so it needs to be changed.

4.3.6 Design Trade-offs

Several design trade-offs have been made, which impact the performance or scalability of SCRIPT. One major trade-off of the design is the use of a centralized element. A central element could reduce performance and could decrease the reliability of the solution. However, the decision to use a centralized element for some tasks was made due to the fact that using this approach a lower latency can be achieved compared to the same tasks being implemented fully in a distributed manner. The load on the *CCR* is expected to be small, since it is used only for management operations, such as identity provisioning, template management, or peer configuration. A node only contacts the *CCR*, when it is started (to receive an identity), when it receives an IPFIX record with an unknown template ID (to receive the template definition, its GTID, and routing function), and when it receives a new template definition (to map the newly observed template to an existing GTID).

Another design trade-off was concerned with the responsibility of the peer awareness task. In the current prototype (cf. Section 4.4), the *CCR* periodically checks the node‘s availability and informs other nodes, when a node becomes unavailable. Designing peer awareness centrally allows for much faster reactions in case of a node being disconnected. A deployment of the solution presented here will not see more than several hundred nodes, thus, such a monitoring task can be performed easily by a single entity due to a reduced number of messages that the *CCR* has to process.

Finally, robustness of SCRIPT can be improved straight forward by adding a secondary *CCR*, which mirrors the configuration and operation of the primary *CCR* and which takes over if the primary *CCR* becomes unavailable.

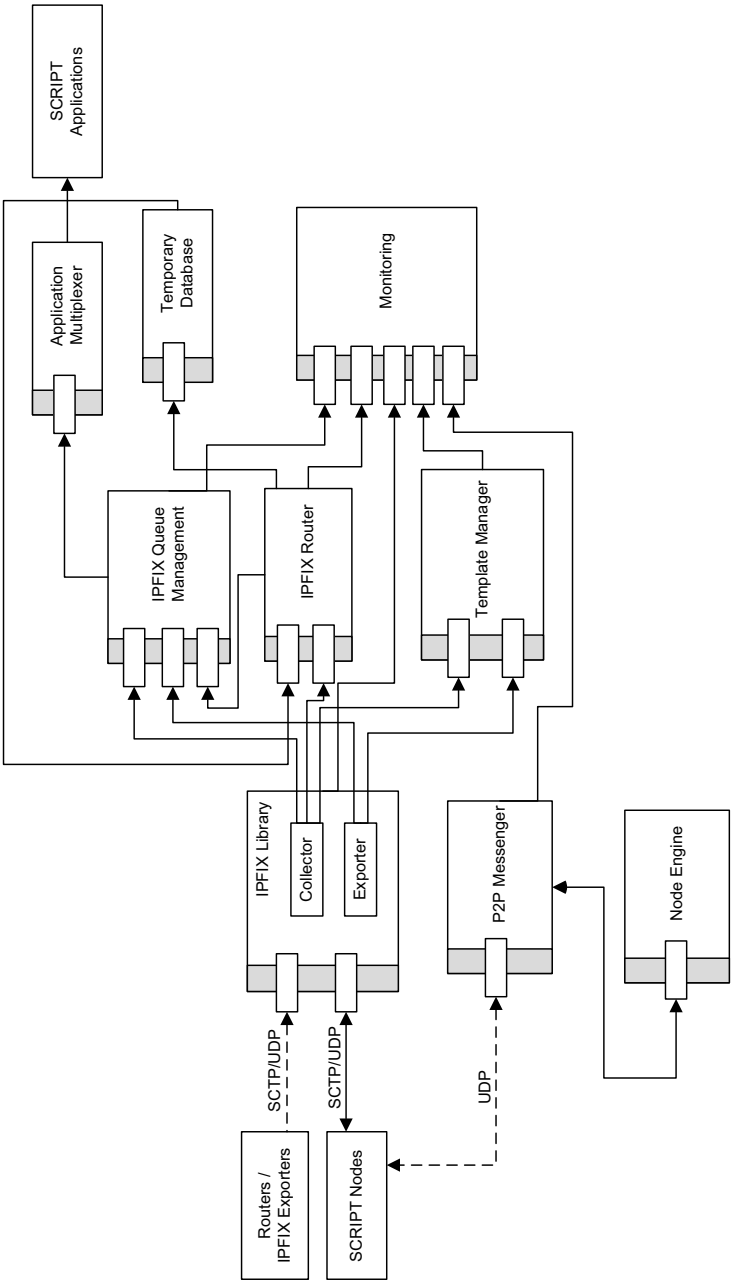


Figure 4.10: SCRIPT Implementation Architecture

4.4 Implementation

Based on SCRIPT network architecture and the internal SCRIPT node architecture presented in Figure 4.9, the implementation architecture is refined and summarized in Figure 4.10. The figure shows the major modules and their interactions. As it can be seen in the figure, SCRIPT uses SCTP and UDP protocols to communicate with external entities. It uses UDP for the P2P overlay management and SCTP or UDP (depending on a configuration parameter) to receive and send IPFIX records. The following sections describe in more detail the most important modules in SCRIPT with the use of class diagrams which also highlight the most important methods which implement the behavior of SCRIPT.

The implementation of the SCRIPT prototype and example SCRIPT applications was done in C++. In the implementation, the Adaptive Communication Environment (ACE) [1] and libsctp libraries were used. The implemented prototype was tested on Linux PCs as well as on Cisco Application Extension Platform (AXP) [17] cards running Linux. Due to the limited support for SCTP on the AXP cards the prototype running on those cards only supports IPFIX transport over the UDP protocol.

As the evaluation results in Section 7.2.1 the implemented prototype achieves its functional goals, and they also show that by deploying a SCRIPT network with several nodes the performance of traffic analysis applications is improved.

4.4.1 P2P Framework

As mentioned above, SCRIPT nodes are organized in a P2P overlay using a communication protocol very similar to the Kademlia protocol. The class diagram representing the software modules used to implement the P2P communication framework are shown in. Figure 4.11

The *P2PNode* class contains a representation of a SCRIPT Node, local or remote. The *LocalNode* is a specialized type of *P2PNode* which besides the properties of a node also aggregates several components. The *Messenger*

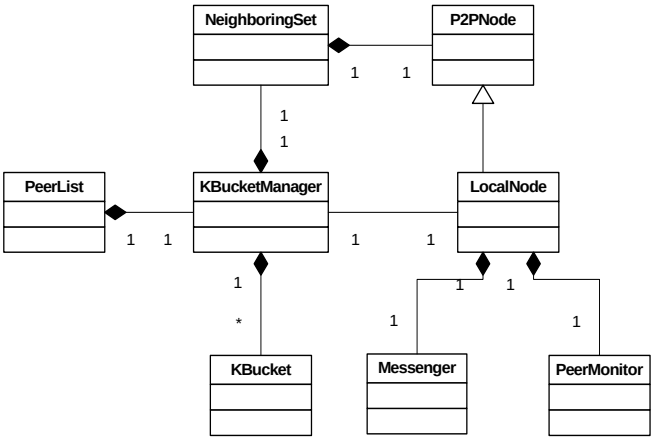


Figure 4.11: P2P Framework Class Diagram

class contains the implementation of the communication protocol used to build and maintain the P2P overlay. The *P2PMonitor* implements a keep-alive mechanism which allows to detect removal of existing nodes, but it is also used to update the P2P routing information. A *KBucketManager* is used to implement the P2P routing algorithm. The *NeighboringSet* class implements a caching mechanism used to store the closest SCRIPT nodes to the current node. This caching mechanism allows SCRIPT nodes to directly communicate with up to *K* nodes which are closest. In case of small SCRIPT networks the caching mechanism makes the overlay behave similarly to a full-mesh network with all nodes being able to communicate directly with each other.

4.4.2 IPFIX Collector

The class diagram representing the SCRIPT IPFIX collector is shown in Figure 4.12. As it can be seen in the figure the IPFIX Collector component is implemented for UDP (User Datagram Protocol) and SCTP (Stream Control Transmission Protocol) which are the two protocols favoured by IETF for IPFIX. The SCTP version, besides using the protocol preferred by IETF, allows better peer-awareness by using SCTP notifications when the status of an association between two nodes changes. When using SCTP, each time a node leaves SCRIPT all the nodes to which that node has an IPFIX session

are immediately notified about the leave, so they can update their routing rules. When UDP is used the availability or not of a remote collector is maintained using keepalive messages in the P2P layer. The most important methods used to instantiate a collector are highlighted in the figure and described below.

The port number used by the collector is passed in the collector's constructor. Once a collector is instantiated, it is started using the *start()* method. Each collector runs in its own thread. Multi thread support is realized by inheriting the *ACE_Task_Base* class of the ACE library and calling the *activate()* method of this class. A collector only decapsulates IPFIX or NetFlow records from IPFIX or NetFlow packets and delivers them to a *Flowhandler* which is registered with the collector by using the *registerFlowHandler()* method. The *removeNode()* method is used by the *SCTPCollector* to inform other components about the drop of the SCTP association with another SCRIPT node.

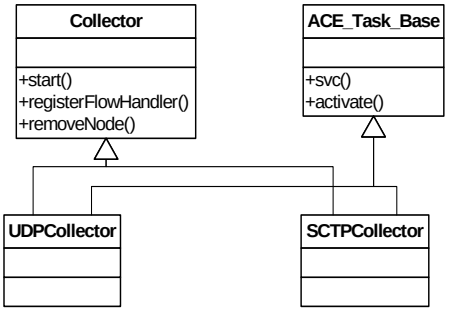


Figure 4.12: IPFIX Collector

4.4.3 IPFIX Exporter

The IPFIX Exporter component layout is shown in Figure 4.13. Similarly to the collector component the *Exporter* class implements the common behavior to all types of exporters, while the *UDPExporter* and *SCTPExporter* classes implement the specifics of the respective transport protocols.

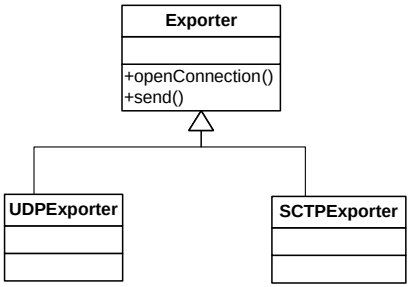


Figure 4.13: IPFIX Exporter

The two main routines of the *Exporter* class are *openConnection()*, which starts an exporting session towards another SCRIPT node, and *send()*, which sends a list of IPFIX records using one of the existing exporting sessions. The list of records as well as the destination SCRIPT node are passed by external components.

4.4.4 Flow Template Manager

Each SCRIPT node instantiates a local *FlowTemplateManager* object which has two main tasks: i) to provide detailed information about a given template, so that the received records are interpreted correctly, and ii) to maintain a mapping between template IDs exported by routers and the SCRIPT Global Template ID (GTID)

Templates are added to the manager using the *addTemplate()* method. When an IPFIX packet containing IPFIX records is received by the collector it requests the *FlowTemplateManager* for the object describing the template of the received IPFIX records using the *getTemplate()* method. If the *FlowTemplateManager* does not know this template it makes a request to the Central Configuration Repository using the *getTemplate()* method of the

ControllerClient. The received template is then stored by the collector in the *FlowTemplateManager().*

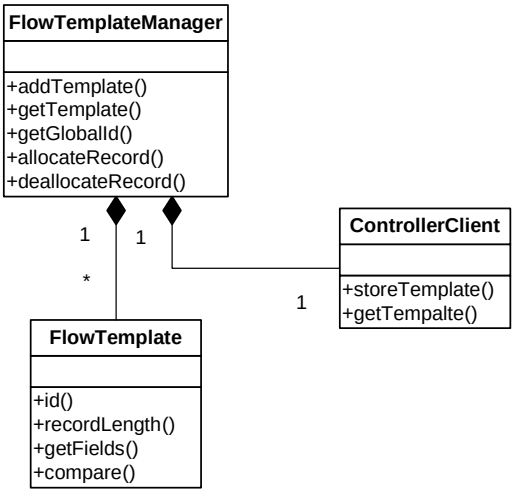


Figure 4.14: Flow Template Manager Class

If the *FlowTemplateManager* knows the respective template then the collector can use the *allocateRecord()* method for each decapsulated IPFIX record to allocate a new data structure specific to that template for storing the record. Once a record passes through all processing stages the data structure used to store it can be released using the *deallocateRecord().*

4.4.5 Flow Records Routing

The flow records routing mechanism is shown in Figure 4.15. The process starts as soon as the *Collector* notifies the *FlowRecordRouter* about a new IPFIX record. Based on the template of the record it then applies a routing function to calculate the *Routing Hash ID* for that record. By passing the calculated hash value to the *SCRIPTEngine* using the *nextRelay()* method a *Node* object is returned which represents the best candidate for processing the respective record given the current state of the SCRIPT network. If the best candidate is the local node then the *deliverLocal()* method is called which passes the record to one or more SCRIPT applications which are waiting for that record.

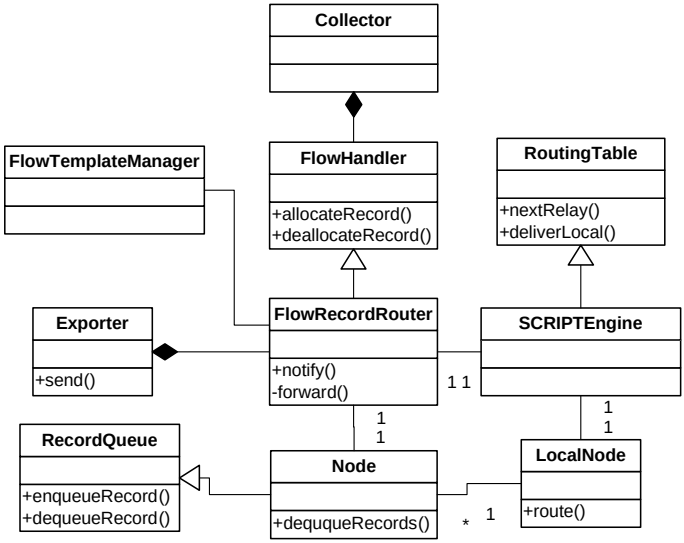


Figure 4.15: Flow Records Routing Class Diagram

If the candidate node is a remote SCRIPT node, then the record is added to a queue of records waiting to be exported to that node. From time to time the queue length of every remote node is checked and if it exceeds a predefined threshold an export process starts.

4.4.6 Application Support

Applications can be built on top of the SCRIPT framework by using an API provided by the framework. An application is started by registering it with the *Controller Unit*, by calling the method *registerApplication(templateId, application)*. The template ID passed in the registration call is used to identify those flow records, which will be passed to the application. If the application needs to receive more than one template, a separate registration call is required for each template ID.

An application needs to extend the class *LocalProcessor* and implement the methods *notify(char *msg)* and *process(sc_flowRecord *fr)*. The *notify* method allows application-specific messages (such as configuration options, or queries) to be sent to an application during runtime. The *process* method

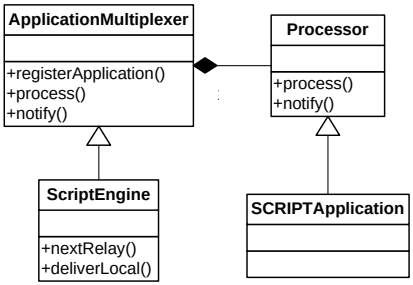


Figure 4.16: Application Support Class Diagram

is called, whenever a flow record with a template ID required by the application is received. A copy of a record is received using the *process* method.

4.4.7 Node Bootstrapping

The bootstrap process represents the action of adding a new SCRIPT node to a SCRIPT network. When starting a node, the IP address and port number of an existing (bootstrap) SCRIPT node is needed for the bootstrap process. During the bootstrap process the new SCRIPT node receives an identity from the *CCR*. For the purpose of identity provisioning, the *CCR* maintains a list of already assigned identities and always tries to assign an identity so that to keep as much as possible a uniform distribution of assigned identities. The algorithm used by the *CCR* for generating a new identity is to find the largest interval between two consecutive assigned identities p and q, and choose the value $N_i = [(p+q)/2]$ as the new identity.

During the bootstrap process besides a new identity, also a set of other existing SCRIPT nodes (IP addresses, port numbers, and node identities) is received from the bootstrap node and these are used to populate an initial routing table. This information is exchanged using the *Control Messaging* component over UDP (User Datagram Protocol) messages. Whenever a node learns about another node in the network, two IPFIX sessions are created, one in each direction, between the *IPFIX exporter* of one node and the *IPFIX collector* of the other node, for exchanging IPFIX records. At the same time the new node is added to the appropriate k-bucket.These operations only take place when a new node connects to the network, so they only create a limited load.

4.4.8 Embedded Environment

Cisco has recently introduced the Application Extension Platform (AXP) [17], which allows applications to run within a router. The implementation of SCRIPT has been compiled using the AXP Source Development Kit (SDK) and tested having two of these nodes running within two AXP cards. The prototype compiled for AXP was based on version 1.1.5 of the AXP SDK. At the time of implementation and testing of the prototype in AXP, the SCTP protocol was not fully supported by the AXP cards, thus, the UDP alternative was applied as a transport protocol for IPFIX, when one or more nodes run on AXP cards. Other than the exclusive use of UDP, the implementation for the AXP is similar to the one for PC.

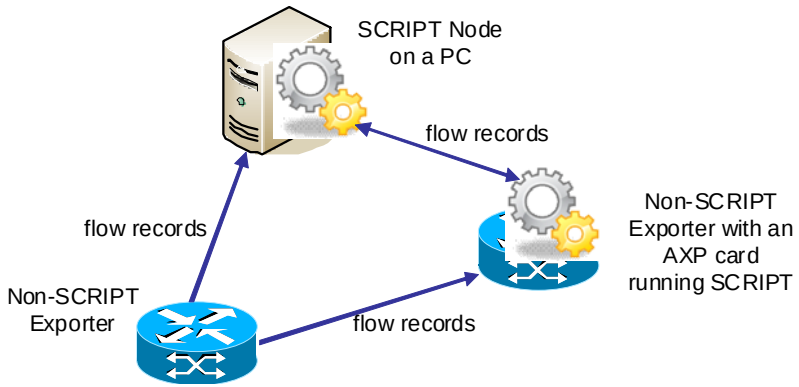


Figure 4.17: SCRIPT Deployment example in a mixed environment with PCs and AXP Cards

The deployment of SCRIPT in a mixed environment with PCs and AXP cards running as SCRIPT nodes is depicted in Figure 4.17.

4.5 Application Integration

As already mentioned, the purpose of the SCRIPT framework is not to solve a particular problem of traffic analysis, but to support development and deployment of a larger set of distributed traffic metering and monitoring applications. Section 3.2.1 introduced three different scenarios for which a distributed system is better suited than a centralized system. This section first

introduces the API that application developers can use to build SCRIPT applications and then evaluates how those three application scenarios can be approached using the SCRIPT framework developed in this thesis. This API allows application developers to register their application with the SCRIPT middleware in order to be notified whenever a record which matches the template required by the application is received. It also allows custom routing policies to be created. The custom routing rules is an advantage, as gives a higher flexibility in handling IPFIX records, but it needs to be used with care. Multiple routing policies for the same template have the effect of multiplying the number of IPFIX records, as whenever an IPFIX record enters a SCRIPT network, a copy of it is injected in the SCRIPT network for every routing policies which matches that template. The default routing policy of SCRIPT considers the 5-tuple flow keys for routing, and the author believes that many application will be able to use it, so custom routing policies will be rather the exception than the rule. All three example applications implemented ontop of SCRIPT make use of the default routing policy.

4.5.1 SCRIPT API

Applications can be developed ontop of SCRIPT by using the API exposed by the SCRIPT middleware. This API allows an application to specify what flow records to use (by passing the template IDs that the application is interested in). The API provides a notification mechanism that informs the application whenever an “interesting” flow record for the application was received by the SCRIPT node on which the application works. In addition, a second notification can be used to exchange application-specific messages over the SCRIPT network.

A SCRIPT node can be started by instantiating a *ScriptEngine* object and passing as parameters a *bootstrap IP address*, *a bootstrap port number*, a *local port number* to be used by the overlay and a port number to be used for the SCRIPT collector component.

In order to receive flow records, a SCRIPT application needs to register to the SCRIPT middleware using the method *ScriptEngine*’s *registerApplication* and pass a *LocalProcessor* object that implements a *notify* method (Figure 4.18). The *notify* method will be called by the SCRIPT middleware whenever a flow record that needs to be processed by the local node is re-

ceived. In addition, the *msgNotify* method can be used if the SCRIPT application needs to receive other type of messages (such as configuration options, or flow queries, etc.).

```
class LocalProcessor
{
   public:
      virtual void notify(FlowRecord *rec);
      virtual void msgNotify (ScriptMessage *m);
};
```

Figure 4.18: LocalProcessor Class

Figure 4.19 shows an example for a flow storage application. It can be seen that the constructor of the application class needs to be called with the application ID assigned for that application. The *setTemplate()* method is used to inform the storage application object about the template of the stored records. In case several templates are used several instances of *StorageApp* objects are required, one for each template. The *createNewFile()* method prepares a new file for storing flow records. It can be called once, when the application instance object is created, or it can be called at regular time intervals in order to have the records distributed in several files based on time. The *writeRecord()* method writes an IPFIX record to the current opened file.

```
class StorageApp: public LocalProcessor,
{
   public:
      StorageApp(uint32_t appId);
      virtual ~StorageApp();
      virtual void notify(char*){}
      void setTemplateId(int tid);
   private:
      void writeRecord(flowRecord *rec);
      int createNewFile(int id);
```

Figure 4.19: Flow Storage Application

In order to deploy a SCRIPT node and use the newly created application, a *ScriptEngine* object needs to be instantiated, and the new application needs to be registered to the ScriptEngine together with the template identifier for the template that the application uses (Figure 4.20). The figure shows two

different applications being started, one instance of the delay measurement application and two instances of the flow storage application. The delay measurement application uses records of template 20001, while the storage application instances will store records of templates 20001 and 20002 respectively.

```
//instantiate applications and assign application IDs
DelayApp *myApp1 = new DelayApp(1000);
StorageApp *myApp2 = new StorageApp(2000);
StorageApp *myApp3 = new StorageApp(3000);
ScriptEngine se();
//register the application with their SCRIPT template IDs
se.registerApplication(20001,myApp1);
se.registerApplication(20001,myApp2);
se.registerApplication(20002,myApp3);
```

Figure 4.20: Application Instantiation

4.5.2 Storage of NetFlow Records

One of the greatest problem network operators have today is the storage and retrieval of flow records. Large operators collect every day Gigabytes of flow records that need to be stored for longer periods of time and may be accessed and analyzed sometimes in future. Handling such amounts of data in a centralized way is costly not only economically, as this requires state of the art hardware, but also in terms of time to access these data. Figure 4.21 shows an example of a traditional centralized collector that is responsible with the storage of NetFlow records originating from three different POPs (Point of Presence) of a network operator. As soon as traffic in the network increases above a threshold, a centralized collector cannot handle anymore the increased load and needs to be replaced with a new, more powerful collector.

Accessing historic NetFlow data is typically time consuming, as this data is often saved in files, and indexes are seldom used. A common practice is to save all flow records within a time frame (*e.g.* 5 minutes interval) in a separate file and stored these files indexed on time. For example, NfSen [78] and flow-tools [47] use this approach for storing NetFlow records. When querying for some particular traffic all flow records from a snapshot are read for each incoming as the data is not indexed. This leads to a query time that

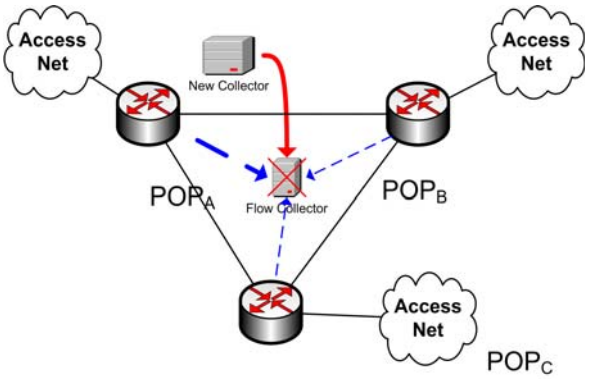


Figure 4.21: Traditional Centralized Collection of NetFlow Data

grows linearly with the size of the snapshot. For complex queries it is often the case that more than a single pass is needed to answer the query, which further increases the query time.

Distribution of NetFlow data to more collectors as depicted in Figure 4.22 allows splitting the workload caused with the storage and retrieval of NetFlow records to multiple machines. As a result, when the system is overloaded new resources (new collectors) can be added in addition to the existing ones and the collection infrastructure can gradually grow with the increase of traffic. The goal of a distributed traffic analysis architecture in this scenario is to combine the storage capacity of multiple nodes, resulting in a larger overall storage space, in order to balance the usage of storage capacities among those nodes, and to achieve redundant storage in order to provide fault tolerance in case of node or network failure, node overload, or network congestion. As the data is distributed to all collectors, each collector stores less data than a centralized collection point, and there are more computational resources to be used for answering queries. This leads to faster query answer times due to less data being parsed by a single node.

As shown in Figure 4.22, routers in the network export flow records to one or multiple collectors that are organized in an overlay collection network. For redundancy purposes more than one collector can be assigned to a single router. The collection network should forward each flow record through the network in such a way that at any point each flow record can be retrieved and at any time the workload is balanced as much as possible between the running collectors.

The content of flow records can include any attributes the operator is interested in. The flow record routing policy of the SCRIPT middleware is capable of routing all flow records to several SCRIPT nodes in order to provide redundancy.

4.5.3 Large-scale One-Way Delay Measurement

Another application that can benefit by distributed processing is the measurement of one-way delay in a large network. In this scenario the aim is to measure one-way delay based on NetFlow records. Delay calculation follows a very simple algorithm: a packet flowing through the network is observed at multiple observation points. Each observation point marks the time when the packet was observed and sends this data to a processing unit. This processing unit then collaborates the data received from all observation points and calculates the respective delays. It is assumed that the clocks of all observation points are synchronized.

Following the proposal in [102], single-packet flows could be use for the delay measurement application. In such a setup, each observation domain is configured to use a hash function during flow creation process such that if a packet is sampled for flow creation in one observation domain, then it will be sampled in all observation domains.

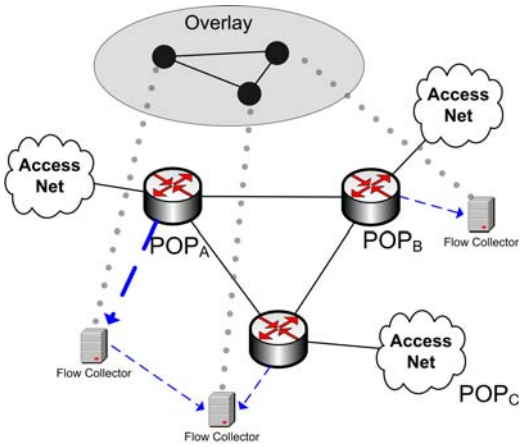


Figure 4.22: Distributed Collection of NetFlow Data

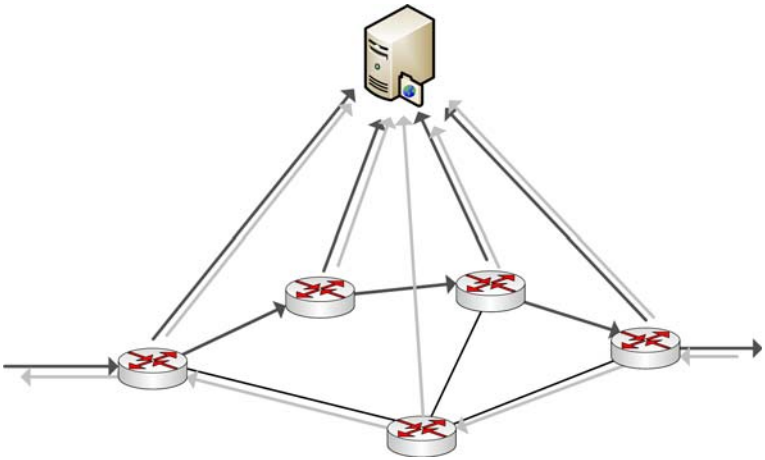


Figure 4.23: Centralized Delay Measurement

In order to be able to calculate the delay close to real-time, flow records generated for the same packet but exported by different routers are routed to the same collector node. This way, each SCRIPT node calculates the delay for a subset of all active flows. In order to calculate this delay, the internal clocks of the exporting routers must be synchronized, and the calculation error for delay depends on the way clock synchronization is maintained. The synchronization of internal clocks is out-of the scope of this thesis.

4.5.4 Asymmetric Route Detection

In this application scenario the SCRIPT platform is used to detect asymmetric routes in an IP network based on flow records collected at different observation points for the two flows corresponding to an end-to-end communication. An asymmetric route is a situation in which a packet does not traverse the same routers in one direction as in the other. For this application scenario is assumed that the clocks of all exporting routers are synchronized.

In this scenario selected routers in the network export flow records to one or multiple SCRIPT nodes, as illustrated in Figure 4.24. Similar to the previous scenario, several SCRIPT nodes can be assigned to a single router for redundancy and load balancing reasons. In order to be able to detect any

asymmetric routes, the SCRIPT middleware forwards flow records belonging to the same flow (in both directions) but exported by different routers to the same SCRIPT node. When this SCRIPT node receives a new flow record, it stores the flow record locally. Records belonging to a flow and its reverse flow are grouped together. After a pre-defined time (this time is required in order to receive all records of a flow from all exporters), the SCRIPT node examines the flow records of the group and compares whether the same routers exported the records in one direction as in the other. The records are ordered according to the start-of-flow timestamp of the record. If the chain of routers in one direction differs from the chain of routers in the other direction, then the route is an asymmetric route.

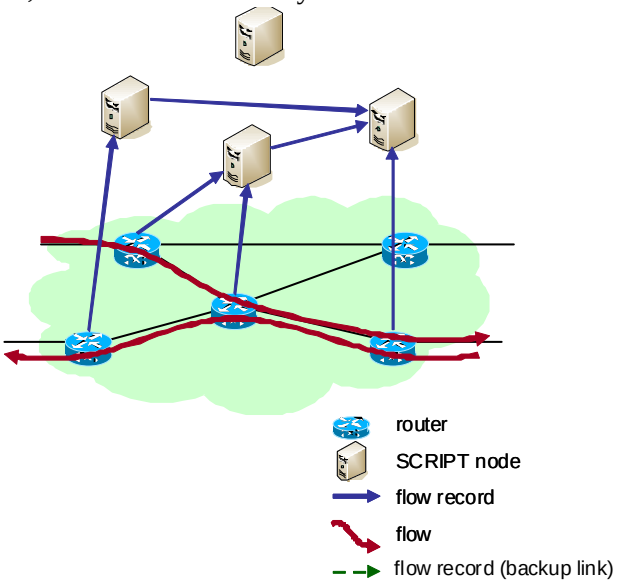


Figure 4.24: Asymmetric route detection scenario

There are different options for the content of flow records. In case of traditional routing (e.g. shortest path first), packet forwarding is performed only based on the destination IP address, and simple metrics such as path costs. In this case, it is sufficient to include source and destination IP address, the timestamp of the first packet of the flow, and the origin exporter into the flow record. In case of policy-based routing, when information from layer 4 and application type are considered in IP routing, the flow record needs to in-

clude additional fields as well, e.g., transport protocol, source and destination ports, and application type.

The flow record routing policy of the SCRIPT middleware needs to route records of a flow and records of its reverse flow to the same SCRIPT node. Therefore, the hash value (Routing Hash ID), based on which the flow record routing is performed in the SCRIPT middleware, needs to be the same for a flow and its reverse flow. Depending on the IP routing policy in the network, this *routing hash ID* value is calculated on different fields. If IP routing is performed based on the destination IP address, the *routing hash ID* value is calculated on the source and destination IP address, so that Hash(source IP, destination IP) = Hash(destination IP, source IP). If policy-based routing is applied, the *routing hash ID* value calculation includes transport protocol, source and destination ports as well.

4.6 Chapter Summary

This chapter introduced SCRIPT, which is a framework for distributed IP traffic analysis. From the generic model for distributed IP traffic metering and analysis introduced in Section 3.4 a dedicated architecture for the traffic monitoring and analysis layer has been derived. Its most important design and implementation details have been presented. SCRIPT is application independent, thus allowing a large set of applications to be developed ontop of it. As examples, a set of three traffic analysis applications using SCRIPT were introduced. As the evaluation results included in Section 7 show, SCRIPT uniformly distributes records to existing analysis machines, and achieves linear scalability. By its design and prototypical implementation, SCRIPT is one possible instance of the distributed monitoring and analysis layer of DITA (c.f. Figure 3.3).

Chapter 5

DiCAP: An Architecture for Distributed Packet Capture

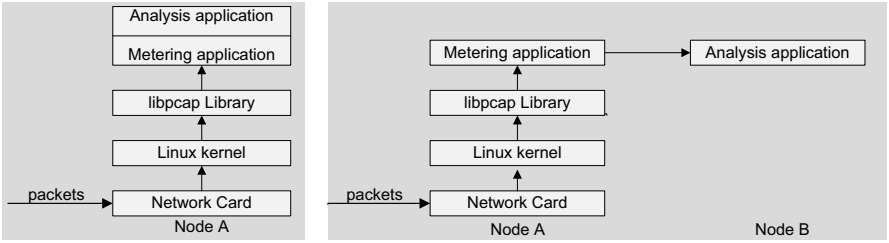
As already described in Section 3.4, besides the distributed traffic analysis component, DITA also requires distributed metering mechanisms to address the problems outlined in the scenario described in Section 3.2.2. The DiCAP [73] architecture developed by this thesis is a distributed traffic metering architecture, which enables software-based traffic metering applications to operate even at high-packet rates, by splitting the packet capture task to several nodes running in parallel. This chapter introduces DiCAP by first presenting the architecture of a typical software-based traffic monitoring applications, followed by an overview of the design and implementation of a DiCAP prototype.

Before sending metered data to one ore more traffic analysis applications (for example a SCRIPT application), metering points need to inspect traffic and extract relevant information. The information extraction from IP packets can be as simple as just increasing a counter of observed packets, or can be more complex including payload inspection, filtering, or aggregation. Software-based packet capturing tools cannot cope with high packet rates and experience high packet losses. This mainly happens because the overhead introduced by the operating system for reading the packet from the network card, and passing it to an application is so high that some packets need to be dropped as the process cannot be sustained at the packet interarrival rate. Hardware solutions to packet capture can accommodate high packet rates, but they are often less flexible and expensive.

In order to address this problem, this section proposes a scalable architecture and its implementation for Distributed Packet Capturing (DiCAP) based on inexpensive off-the-shelf hardware running the Linux operating system. The prototype has been tested and evaluated against other Linux capturing tools. The evaluation shows that DiCAP can perform loss-less IP packet header capture at high-speed packet rates when used alone and that it can improve highly the performance of *libpcap* or *PFRING* when used in combination with those.

Packet capturing performance on high speed network links is highly influenced by the performance of the capturing hardware and software. Two limiting factors for packet capture tools are the memory and CPU speed on the capturing node. The higher the packet rate on the network link, the less time available for capturing and analysis of a single packet. As the evaluation of DiCAP shows, the capturing performance of a Linux machine drops significantly at high packet rates. Such decrease in performance is due to the processing of each packet by the kernel. By parallelizing the packet capture task to several capture nodes, the number of packets captured and analyzed by a single node is decreased, thus the time available to process (analyze) a single packet is higher, so chances of dropping packets due to lack of available resources is reduced.

The strengths of DiCAP are twofold: i) it allows a scalable approach to traffic analysis by enabling the addition of new nodes when the load on the existing nodes increases; and ii) it can be deployed together with already ex-



a) colocated analysis and metering

b) separate metering and analysis

Figure 5.1: Two examples of libpcap-based applications

isting libpcap-based applications in order to make these applications run in a distributed environment.

5.1 Architecture of a Typical libpcap Application

As the default behavior of the operating system running on a network connected device is to drop all packets which do not have as a destination that device, special tools need to be used by traffic analysis applications that need to access all traffic observed on a network link. Most traffic analysis applications built for the Linux operating system use either *libpcap* or *libpcap-PFRING* library for accessing all packets observed by a network interface card.

An application developed using one of these libraries can be seen as a combination of two processes: a metering process and an analysis process. The metering process is responsible with retrieving the relevant data from the observed packets, while the analysis process is responsible with running an algorithm on these data and find out relevant details about the traffic. Figure 5.1 shows two deployment scenarios for *libpcap*-based applications. The first example shows the mostly used application design in which the metering task is collocated with the analysis task. Each packet observed on the network link is delivered to the analysis application, which extracts the relevant data and runs the analysis algorithm on these data. The second example separates the two processes as follows: one libpcap application performs metering based on the observed traffic and sends metering data to an analysis application. The analysis application may be collocated on the same node as

the metering process, but most often runs on a separate PC. An example of such a separation is a flow accounting application in which the metering process creates flow records based on the traffic observed on the link and exports them using a flow exporting protocol to a collector in the network which runs one or more traffic analysis applications on the received flow records.

5.2 DiCAP Design

The main idea of DiCAP is to split the load of packet capturing load between several self-organizing capture nodes placed on the network link, and have each of them "sample" the traffic in such a way that no two capturing nodes capture the same packet. Each capture node receives a copy of the whole traffic and decides locally which traffic to capture and which not. DiCAP allows for scalable packet capturing by increasing the number of capture nodes, when the increase of traffic requires.

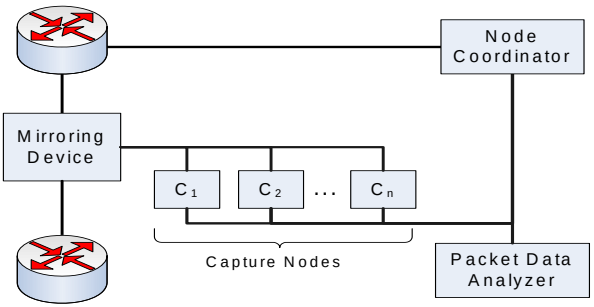


Figure 5.2: DiCAP Architecture

5.2.1 System Architecture

The DiCAP architecture (cf. Figure 5.2) is made of several *capture nodes* organized within a capture cluster. The *capture nodes* are coordinated by a *node coordinator.* The *node coordinator* controls the capturing process and configures the policy used by each *capture node* to decide which packets to capture. The architecture shows a single logical component for the coordination task, but for increased robustness secondary node coordinators could be

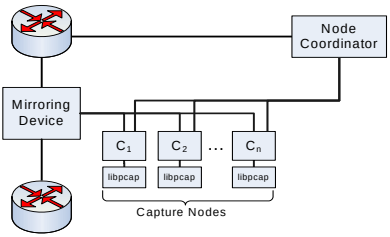


Figure 5.3: DiCAP in Distribution Mode

present in order to take over the coordination in case the main coordinator crashes.

As shown in Figure 5.1 a traffic analysis application can be collocated with the metering process, or separated. In case the analysis application is a separate process, a *packet data analyzer* is introduced in the architecture as a generic representation of an analysis application. In DiCAP, the *packet data analyzer* is used to receive captured packet headers from the capture nodes and further analyze them. Similarly to the node coordinator, the *packet data analyzer* may also be distributed. Besides increase of robustness, distributing the *packet data analyzer* can also be used for load balancing. In case of an analysis application collocated with the metering process, the *packet data analyzer* is not required, and DiCAP can be configured to run in a separate operation mode, called *Distribution Mode* and the DiCAP system architecture slightly changes to the one shown in *Figure 5.3.*

Live traffic observed on a high-speed network link is mirrored to a capturing cluster made of several capturing nodes. Each capture node has two network interfaces: a *passive* network interface on which traffic to be captured is received from the mirrored link and an *active* network interface that is used to communicate with the *node coordinator(s)* and the *packet data analyzer(s).*

5.2.2 Management of Capture Nodes

Each capture node in DiCAP has a 32 bit *nodeID* which is used to uniquely identify a *capture node* within a cluster of capturing nodes. In order to participate in the capturing process, a candidate *capture node* needs to associate

to a *node coordinator* by sending a *join* message which includes the node's identity. Once the node coordinator receives the *join* message it adds the received *nodeID* to its list of known active capture nodes. Additionally, the node coordinator informs the whole capturing cluster about the updated cluster topology using a *synchronization* message. The *node coordinator* has the task of informing capture nodes, which packets they need to capture based on one of the two mechanisms described later in Section 5.2.5. For keeping a consistent view on the topology of the capture network each capture node sends a heartbeat message every 5 seconds to the *node coordinator.* If the *node coordinator* does not receive a heartbeat message for 15 seconds from one of the capture nodes, that *capture node* is assumed offline and it is removed from the active capture nodes.

A capture node learns about a successful join operation if it receives an *accept* message from the *node coordinator.* After being accepted, a capture node will not capture packets until it is included in the capture cluster. A node detects its inclusion in a capture cluster by checking synchronization messages sent by the node coordinator, which include the most current topology of the capture network. If a node find itself in the topology, then it starts the packet capture process.

5.2.3 Logical Topology of the Capture Cluster

The logical topology of the capture cluster can be seen as a bus and is created based on a distributed algorithm coordinated by the *node coordinator.* Each *capture node* decides which packets it has to capture based on the rules received from the *node coordinator.* Two different mechanisms for packet selection are proposed by DiCAP: *round-robin selection* and *hash-based selection.*

In the *round-robin selection* capturing nodes are logically organized in a circular list. From synchronization messages received from a network coordinator each node knows how many active capture nodes are in that circular list and also knows its own position in the list. For each incoming packet one single node is designed as responsible to capture (and eventually process) that packet. That node captures the packet, while the next node in the circular list becomes the responsible capture node for the next packet on the link.

In *hash-based selection* the capturing nodes form a Distributed Hash Table (DHT) so that each node is responsible with a particular range of hash values. The responsibility assignment is made by the node coordinator and sent from time to time to each capture node using synchronization messages. For every incoming packet each capturing node applies a deterministic hash function on a subset of the packet header fields and captures the packet if it is responsible for the resulting hash value. A more detailed explanation of the two selection mechanisms is given in Section 5.2.5.

The packet selection algorithm highly impacts the way the capture load is distributed among the *capture nodes.* The *round-robin selection* allows for a perfect distribution of workload, but it requires injection of control packets by the *node coordinator* into the monitored traffic. The *hash-based selection* can be totally passive but it has two disadvantages:

- A hash function to always distribute packets equally is difficult to find.
- Calculating a hash for each packet is more computational-intensive than just increasing a counter (as in *round-robin selection*)

5.2.4 Control Messages

The type of selection to be used as well as other management information is communicated to the *capture nodes* by the *node coordinator* by using synchronization messages. Based on the different message types exchanged between *capture nodes* and a *node coordinator* a generic control message format is defined and shown in Figure 5.4.

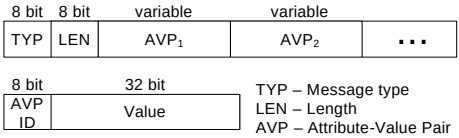


Figure 5.4: Control Message

Each control message contains a type (TYP) field which specifies the control message type. Three types of control messages have been defined for DiCAP: *join, accept,* and *synchronization*. The information in the control message is organized in AVPs (Attribute-Value-Pairs). Just after the type

field of the control message the length (LEN) field specifies how many AVPs are contained within the control message. Figure 5.4 also shows the format of an AVP. An 8 bit code specifies the attribute type and it is followed by a 32 bit value for that attribute.

Figure 5.5 shows the different type of AVPs defined for DiCAP. The *Node ID* AVP is used in all three control messages. In the join message it specifies the new *capture node* identifier. In the accept message it specifies the intended capture node for which the accept is issued. Also in the accept message the *Coordinator IP* and *Coordinator Port* AVPs are used to inform about which IP address and which port number the synchronization messages will use as a source. Based on these, a *capture node* identifies synchronization messages in a stream of traffic packets. The *Analyzer IP* and *Analyzer Port* specify the IP address and the port number where captured data should be relayed to for further analysis. The *Selection Type* AVP specifies which type of selection to be used.

In the *topology update* message a list of this AVP is used to define the topology of the capture cluster.

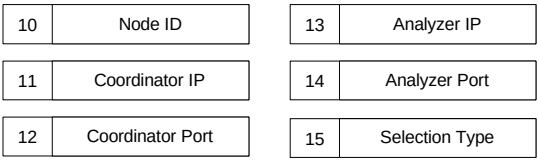


Figure 5.5: DiCAP AVPs

5.2.5 Packet Selection Strategies

As already introduced in Section 5.2.3 two different packet selection strategies are proposed by DiCAP: *round-robin selection* and *hash-based selection.* Regardless of the selection type used the *node coordinator* maintains a list of *nodeIDs* of active *capture nodes*. Each of those *nodeIDs* is kept in for as long as the *node coordinator* receives heartbeat messages from the respective *capture node*. Using synchronization messages the node coordinator sends regular updates to announce the topology of the capture cluster. Based on the topology and the selection method each node can decide which packets to capture.

In the *round-robin selection* synchronization messages are sent in-line as packets are injected in the monitored traffic every *k* seconds. Additionally, a synchronization message is sent whenever the *node coordinator* learns about a new active *capture node*, or detects that a *capture node* is down. The synchronization message contains a list of *nodeIDs* of all active *capture nodes*. Upon the receipt of such an update message each capture node learns about the total number of capture nodes (*N)* and it's current position *(P_i)* in the received list of capture nodes. A packet counter *C* is reset to 0 by the receipt of a control packet. The packet counter is incremented with each packet seen on the wire. A capture node captures a packet if after the receipt of the packet the following equation holds true:

$$C \bmod N = P_i - 1$$

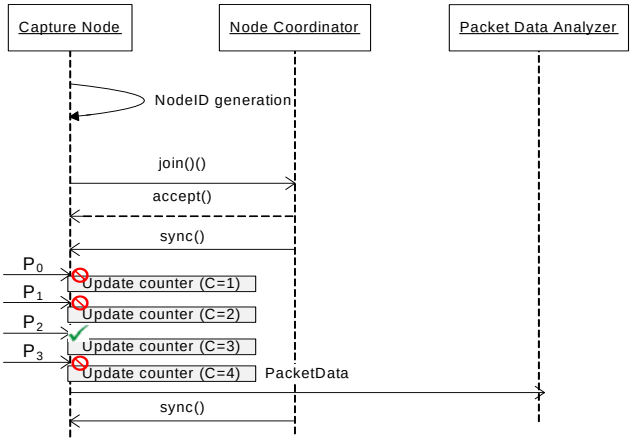


Figure 5.6: DiCAP Communication

The mechanism is also depicted in Figure 5.6. First, a new capture node is deployed. After generating its *nodeID* a *join* message is sent to the *node coordinator* which accepts the new capture node. After a while a synchronization message is sent with a new cluster topology. The example assumes that the newly created *capture node* is assigned the third position in the topology of the capture cluster. Once it receives a synchronization message it resets counter *C* to *0*. The figure shows that the first two packets are dropped (as the corresponding counter values are 0 and 1) while the third one is captured (for *C*=2). The counter shall be again set to *0* upon the receipt of the next topology update message.

In order for this mechanism to work it is required that each capture node receives all packets in the same order. The mirroring device needs to ensure that on all mirrored ports the traffic is sent in the same order.

If such a system that assures that all capturers receive all packets in the same order cannot be satisfied, the decision to capture a packet is taken based on a deterministic hash function applied on the header of the IP packet. In this case, each *capture node* captures the packet if and only if the result of the applied hash function is a value for which the node is responsible.

As the hash values determine which node captures a particular packet, in order to fairly distribute the workload between the different *capture nodes* in this second approach, the hash values generated by the chosen hash function need to follow as close as possible a uniform distribution function. One obvious choice is to apply the hash function on the IP address and port number fields in the IP header, but that has the big disadvantage that in situations in which an increased amount of traffic flows to/from the same IP address/port number (such as large data flows or denial-of-service attacks) the node responsible for the involved IP address or port number gets overloaded. In order to avoid such situations DiCAP uses a hash function applied on the identifier field of the IP packet header. The identification field contains a 16 bit value that is used to identify the fragments of one datagram from those of another. The IP protocol specifications requires that the value must be unique for a source-destination pair and protocol, for as long as the datagram will be active in the Internet system. Supposing the identifier value for a packet is *I*, *N* active *capture nodes* are available, and the position of the current *capture node* in the *capture node* list is P_i, the packet is captured if and only if:

$$I \bmod N = P_i$$

The main advantage of this approach is that synchronization between the capturers does not have to be as tight as in the round-robin approach. In this case the capturers only need to have synchronized the topology, but the order of packets they see on the link does not have to be identical. A disadvantage however is the use of the identifier field which does not always lead to an equal distribution of capture responsibility among the different capturers. During the evaluation of DiCAP several tests have been performed to see how good the use of the identifier field is for balancing the load. The results of those tests are detailed in Section 7.2.2.

Although the implemented prototype uses a hash function applied on the IP identifier field, other implementations may use any type of hash function applied on any part of an IP packet.

5.2.6 Operation Modes

The design of DiCAP allows for two modes of operation: a *distributed metering mode* and a *distributed analysis mode*. The *distributed metering mode* strictly follows the architecture described in Figure 5.2 while the *distributed analysis* mode is depicted in Figure 5.3. In *distributed metering mode* each capture node retrieves packet headers from the monitored link and sends those headers to a *packet data analyzer.* The main drawback of this approach is that only the information packet header is collected and used in the analysis application, while the payload is dropped.

The *distributed analysis mode* addresses the above problem and allows analysis applications access the whole content of a packet. In this operation mode an instance of the traffic analysis application runs on every capture node, thus a *packet data analyzer* is not required anymore. Instead, when a capture node needs to capture a packet, it delivers that packet to the analysis application instance running locally.

5.3 DiCAP Module Implementation

A prototype following the DiCAP design was implemented in C and C++ under Linux. It consists of a Linux kernel module called DiCAP that runs on each capture node, a node coordinator that is used to synchronize the capture nodes, and a packet data analyzer which collects all the packet headers captured by all capture nodes. For the implementation of DiCAP module prototype, version 2.6.16 of the Linux kernel was chosen.

5.3.1 Implementation Architecture

In order to avoid problems observed in capturing packets at high packet rates with libpcap, the major bottlenecks which degrade the behavior of libpcap have been identified. During early investigations of causes which lead to

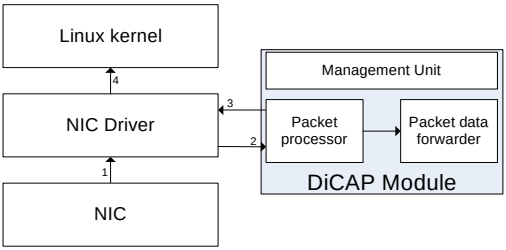


Figure 5.7: DiCAP Kernel Module

a drop of performance of libpcap it was clear that capture problems that appeared while capturing packets, high packet rates were caused by the Linux kernel, which performs several tasks (such as memory allocation, copy of memory blocks) on each captured packet. In order to maximize the performance, the DiCAP kernel module was chosen to be implemented as low as possible in the kernel networking stack in order for packets that should be dropped to be excluded as early as possible from the packet handling done by the Linux kernel. The architecture of a DiCAP-enabled Linux kernel is depicted in Figure 5.7.

Traditionally, each packet captured by the network interface card (NIC) is forwarded to the NIC driver which allocates a memory structure (*sk_buff*) for the packet, copies the packet data in this memory structure and then forwards the newly created *sk_buff* to the Linux kernel. DiCAP module is placed within the NIC driver so that packets that should not be captured by a capture node are dropped before their corresponding *sk_buff* structure is created. Similar architectural approaches have been proposed for other network monitoring projects such as [32] in order to achieve fast processing of network packets. Having the DiCAP module called by the NIC driver makes the implementation device-dependent, but the modification of the driver only requires a function to be called before the respective *sk_buff* structure is created, so it can be easily ported to any other NIC driver.

As Figure 5.7 shows, the DiCAP module consists of three main components: a *packet processor*, a *packet data forwarder*, and a *management unit*. According to Section 5.2.6 DiCAP can be used in two modes: a *distribution capture mode* and a *distribution mode*. The implementation details for the two modes are further detailed in the following subsections.

As the systems used to test and evaluate the DiCAP prototype had a Broadcom [12] Gigabit Ethernet network card, the Broadcom Tigon3 ethernet driver was modified in order to support DiCAP.

5.3.2 Module Implementation

DiCAP runs as a Linux kernel module and instantiates an own kernel thread which is used to send metering data to a *packet data analyzer*. The DiCAP module is called by the receive routine of the network interface card driver. The prototype used a Broadcom Gigabit card, so the respective function for kernel 2.6.16 is located in the file */drivers/net/tg3.c* and its signature is:

```
static int tg3_rx(struct tg3 *tp, int budget){}
```

For each incoming packet a *sk_buff* structure is allocated and associated to the content of the packet. If the DiCAP module is loaded then the skb structure is passed to the module by calling the function:

```
int dicap_process(sk_buff *skb);
```

If *dicap_process* function returns 1, then the respective packet is allowed to be processed further by the Linux kernel and other user-space applications. If the return value is 0, then the respective packet is immediately dropped by the kernel. When DiCAP operates in *distributed metering mode*, the *dicap_process* function always returns 0, as the metering data is extracted by the DiCAP module itself. When DiCAP operates in *distributed analysis mode* the DiCAP module only selects which packets are allowed to be passed to a user-space libpcap-based application.

The *dicap_process* routine first checks if a packet received is a control packet sent by the node coordinator. If so, then it reads the payload of the packet and configures itself accordingly. Control packets have the format specified in Figure 5.4. Always control packets are dropped after processing and they are not allowed to be further processed by the Linux kernel.

5.3.3 Distributed Capture Mode

In the *distributed capture mode* the DiCAP kernel module distributively captures IP packet headers and sends them to a packet data analyzer.

The *packet processor* has the task of deciding whether a packet is going to be captured or not. It receives a packet from the NIC and first it checks on which interface the packet was received. If it was not received on the monitoring interface the *packet processor* does not handle it and informs the NIC driver to deliver the packet to the Linux kernel. If the packet was received on the monitoring interface, based on one of the rules as described in Section 5.2.6, it will do one of the following:

- If the packet is to be captured locally the packet data (in the prototype version the packet header) is sent to the *packet data forwarder* while the NIC driver is instructed to drop the packet.
- If the packet is not within the responsibility of the current packet capturer, the NIC driver is instructed to drop the packet.

The *packet data forwarder* has the task of sending the captured data to a *packet analyzer.* In the implemented prototype it stores in a buffer the first 52 bytes of each packet, including the IP and transport layer headers. Once the buffer is filled it is sent via UDP to a *packet data analyzer.* In the current implementation in each packet sent to the *packet data analzyer* there are 28 packet headers.

The *management unit* performs maintenance tasks. It is implemented as a separate kernel thread. The *management unit* maintains communication with the node coordinator and gets from time to time the current topology of the capturing network as well as information about the address (or addresses) of packet analyzers.

5.3.4 Distribution Mode

In *distribution mode* the DiCAP kernel module has the task of distributing the packet capturing task between several nodes. Packet capture does not take place in the module but in higher level libraries such as *libpcap*. As a result the *packet forwarder* is not used in this mode.

While the *management unit* has a similar task as in the *distributed capture mode*, the *packet processor* has a slightly different behavior. For each incoming packet, the *packet processor* decides whether it is going to be locally processed or not. If the packet is going to be locally processed the NIC driver is informed to forward the packet to the Linux kernel. Otherwise the NIC driver is informed to drop the packet.

5.4 Chapter Summary

This section presented the design and prototypical implementation of DiCAP, an architecture for distributed IP packet capturing. DiCAP is a mechanism of the distributed metering layer of DITA (c.f. Figure 3.3). It does not require any dedicated hardware, which makes it a cost-effective solution for capturing IP packet headers at high packet rates. As the evaluation results show in Section 7, DiCAP experiences no loss at high packet rates, whereas libpcap and libpcap-RING experience up to 96% packet loss at the same packet rates. An important characteristic of DiCAP is the possibility of choosing one of two operation modes: *distributed capture mode* and *distributed analysis mode*. In *distributed capture mode*, DiCAP collects packet headers from all packets on a network link and sends them to one or more analysis nodes. If packet analysis is performed only on information contained in the packet headers, then the *distributed capture node* is the better choice, as the packet header reading is done in a very efficient way, and a single node can retrieve packet headers for links with more than 500.000 packets per second. In *distributed analysis mode*, DiCAP can be used in parallel with other packet capturing tools, such as libpcap or libpcap_PFRING, in order to increase their performance, by distributing their workload across several nodes. In this mode, DiCAP does not capture anything, but controls which packets should be capture by other capture libraries. This mode is better suited for already existing applications in order to run them distributed. Being implemented as a LINUX open-source project, DiCAP can easily be extended with further functionality. The scalable architecture of DiCAP allows network professionals dealing with network measurements to increase strongly the performance of their measurements by adding new resources into the packet capturing infrastructure. The simplicity of the design allows DiCAP to be easily implemented in hardware, leading towards a hardware dedicated packet capture cluster architecture.

Chapter 6

LINUBIA: Linux-supported User-based IP Accounting

In order to be able to map IP packets to users and processes, rather than end devices, as required in Section 3.3, a dedicated metering mechanism which follows the DITA architecture is needed. The *Linux-supported User-based IP Accounting* (Linubia) [75] mechanism developed in this thesis, and presented in this chapter, allows for very granular IP traffic accounting on Linux based hosts connected to the Internet. First, an overview on terminal computing as one example of multiple users sharing the same network end-device is given. Then the design of the Linubia accounting module is given followed by the most important implementation details for the prototype implemented for the Linux kernel version 2.6.

Obtaining information about the usage of network resources by individual users forms the basis for establishing network billing systems or network management operations. Such systems cannot be reliably built and deployed if a particular traffic or IP flow cannot be mapped to an identifiable end user. While there are already widely used mechanisms for metering IP network traffic on a per-host basis, there is no adequate solution for accounting per-user, or even per-process network activities on a multi-user operating system. This is due to the traditional approach to IP traffic monitoring which assumes one or several metering points in the access or core network of an operator, but does not include a measurement point in the end-hosts connected to the network. By measuring traffic directly on the end-hosts, Linubia allows collection of very granular information about the user and application which sends or receives each packet, by looking at the process which sends or receives IP data. This helps network administrators who want a detailed control on network usage based on user policies, rather than host (IP) policies.

In the following sections the architecture of Linubia is introduced, and a prototypical implementation for Linux operating systems is proposed. The implemented prototype is capable of providing per-user accounting for IPv4 and IPv6 protocols. The evaluation of Linubia (c.f. Section 7.1.3) shows that the extra processing required to collect accounting information in the end-hosts is very small, thus it does not impact the performance of IP transfers.

6.1 Terminal Computing

Section 3.2.3 already introduced three different scenarios in which a traffic accounting system capable of matching a particular packet to an user or application is desired. Such scenarios are specific to terminal computing in which multiple users access a *terminal server* in order to use its resources.

A terminal server is a network connected machine, hosting a multi-user operating system to which users having an account can connect and run their applications. Terminal server environments are not so common today as they were more than a decade ago. As a typical PC became more and more powerful, the need of terminal servers grew smaller. However, during the last few years more and more mobile devices have been introduced which allow users to access the Internet from virtually anywhere. In this context it may be that

the terminal server paradigm will gain popularity again (in one form or another - for example as cloud computing), as mobile devices do not have the required resources to perform very complex jobs, so users would rely on remote machines to run the most demanding applications. Most of the time Internet users use their own devices (such as mobile phones, laptops, home PCs) for connecting to the Internet, but sometimes due to special requirements (for example running a long-lasting job, or high bandwidth connectivity) the access to a terminal server is desired. For the remainder of the chapter, in order to ease understanding, a terminal computing environment is assumed. However, the solution developed here is applicable in the same way in a cloud-computing environment, in which multiple users share computational and network resources.

Figure 6.1 shows a traditional architecture for a terminal computing environment. It represents multiple users (labelled user 1-3) which use terminals (labelled terminal 1-3) to connect to a terminal server. On the terminal server the user accounts are locally managed and multiple users can work concurrently on the same system. The users inside the terminal server are dashed to show they do not have to be physically in front of the machine. Now each of those users can use the terminal server for starting and running network applications.

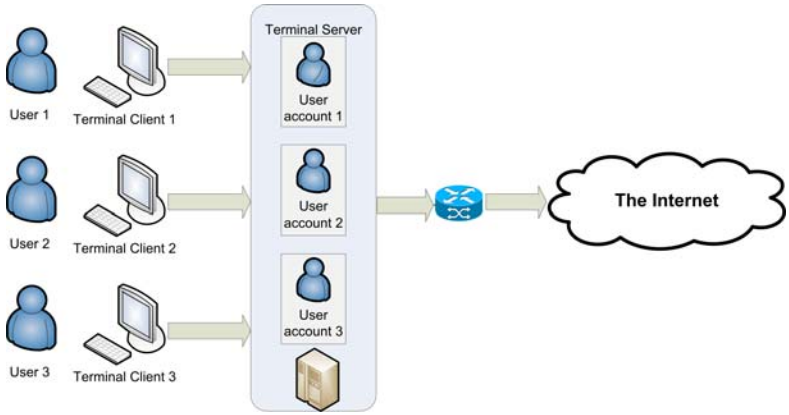


Figure 6.1: A pure terminal environment [44]

The typical scenario covered by LINUBIA goes beyond a singular terminal server and assumes an enterprise network consisting of several multi-user

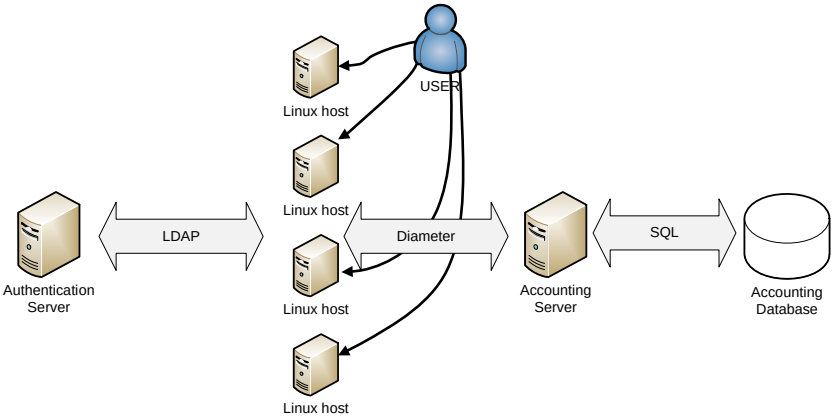


Figure 6.2: Enterprise Network Architecture with Linubia

operating systems that are accessed by anyone who has its credentials registered with the enterprise authentication and authorization platform.

6.2 User-Based Accounting Architecture Design

The architecture of an enterprise-wide network having LINUBIA deployed on its Linux end-hosts, consists of both a network architecture (cf. Figure 6.2) that defines the network components required for LINUBIA and a end-host architecture (cf. Figure 6.3) that defines the software components required within an end-system to support user-based IP traffic accounting

.There are two types of devices that can be identified in the network architecture: regular Linux hosts, which are used by users for running their network-intensive applications, and service enabler devices (which are not directly accessible by end-users) for building the AAA domain infrastructure (an authentication and authorization server, an accounting data aggregation server, and a storage database for accounting records) which collects accounting data.

Linux nodes use the AAA domain infrastructure in order to authenticate their user credentials at an authentication server, and obtain authorization privileges for using resources on the regular Linux hosts. The regular Linux hosts use an accounting server to send accounting records describing the net-

work usage of each individual users, while the storage server is used by the accounting server to store all accounting records received from the Linux hosts. Example of authentication servers in such environments are Light-weight Directory Access Protocol (LDAP) [89], or Kerberos. Accounting functionality is traditionally achieved with the Radius or Diameter protocols. Due to its flexibility, IPFIX could also be used to transport accounting data from the Linux hosts as IPFIX records.

Whenever a user logs in to a Linux host all the processes started by the user will run with the global UID of the user. Each Linux host has LINUBIA enabled. Accounted for data is encapsulated in accounting records and it is transported from each Linux host to an accounting server using the Diameter protocol. The accounting server further stores the accounting records in a central database. For supporting this an accounting client runs on each host, collects the data accounted by LINUBIA and sends it to an accounting server using the Diameter protocol.

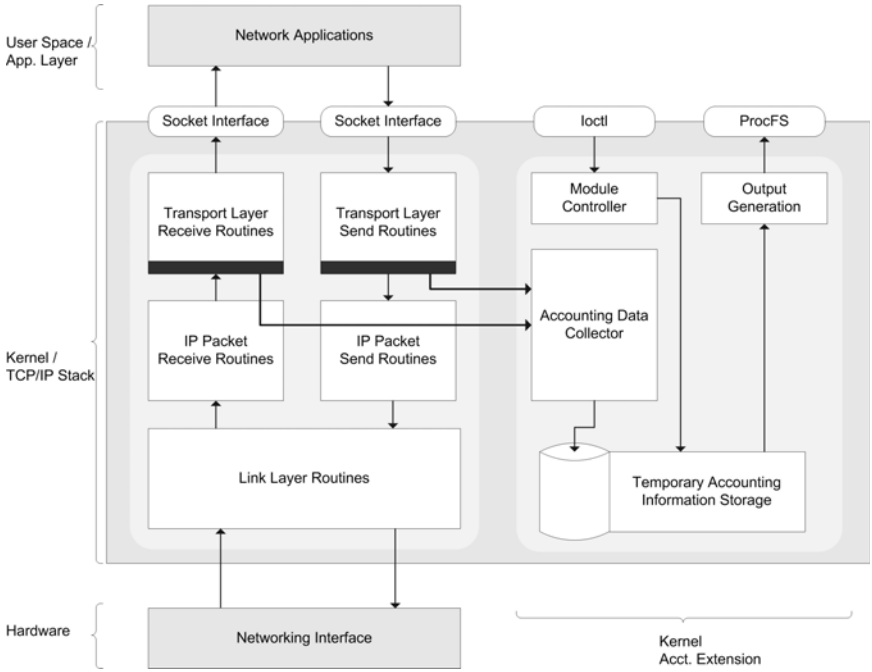


Figure 6.3: End-Host Architecture

6.2.1 End-Host Accounting Architecture

The core element of LINUBIA is the end-host architecture, which performs the actual traffic metering process. Regular operating systems do not offer a method to autonomously measure user-specific or application-specific IP traffic. Therefore, the network stack of the operating system running on a host needs to be modified in order to be able to perform such a task. Figure 6.3 shows how this is achieved by modifying the Linux kernel which is responsible with delivering data between user-space applications and active network sockets. The Linux kernel allows network applications to access the TCP/IP stack via the network socket interface which contains routines for sending outgoing TCP or UDP segments to the network and to receive incoming packets in user-space applications. These routines and the kernel have to be extended in order to measure, store, and export the desired accounting information associated with each accounting-relevant IP network operation. This is done by a kernel accounting extension that consists of a number of components, shown on the right side of Figure 6.3, which are added to the kernel.

The *information storage* component is responsible for the temporary storage of accounting information collected. It resides in the main memory and it resets at every restart of the operating system. For each user which is active (has a running process which creates network traffic) it keeps several integer values which describe the network usage of that user (for example number of bytes and packets sent and received for IPv4 and IPv6 traffic).

Each packet flowing from an application to a socket, or from the socket to an application triggers a lookup in this component for finding the record entry corresponding to the user responsible with the respective traffic thus the efficiency of the *information storage* component highly impacts the overall performance of the accounting module. The *accounting data collector* component is responsible with mapping a particular data transfer (incoming or outgoing) to an user or a process. It extracts this information from the IP networking subsystem and adds it to the storage component following the process described in Figure 6.4.

The *output generation* component is a presentation component which formats the internal data before exporting it to user space via the proc file sys-

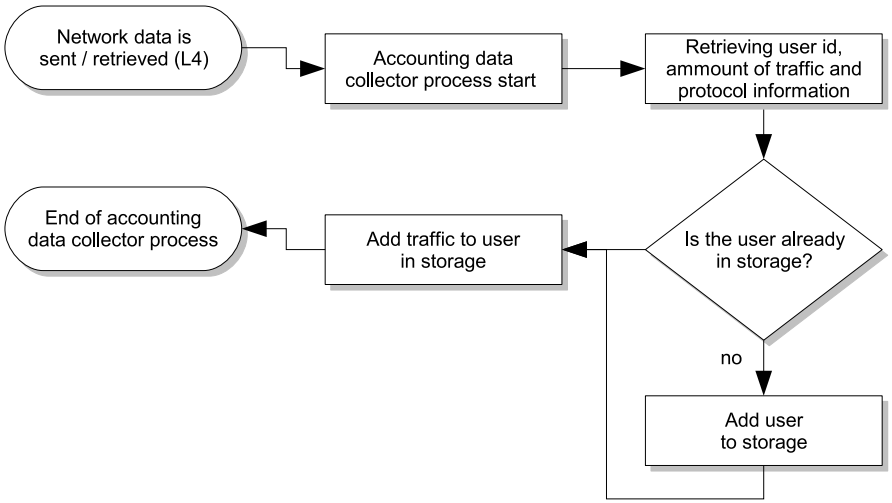


Figure 6.4: Accounting client collector process

tem (`procfs`). The *module controller* provides facilities to manage records stored, for example to reset all records of a specific user. The module controller is accessed using the `ioctl` interface.

This end-node architecture is designed to extract at the kernel level relevant user-specific IP accounting data from the information exchange between user-space applications and network sockets, and export these data to other more-complex IP traffic accounting applications running in the user-space. Such accounting applications could be a Diameter or Radius accounting client, an IPFIX exporting process, a local monitoring application, or a simple permanent accounting storage application that extends the functionality of the temporary storage component.

6.2.2 Linubia Components View

Figure 6.5 shows all the components used by Linubia and the integration of the host specific architecture into the network architecture. As it can be seen in the figure, two protocols make the link between the AAA infrastructure and an end node:

- A protocol for user authentication and authorization of resource usage (such as LDAP, Kerberos, Diameter, etc.)

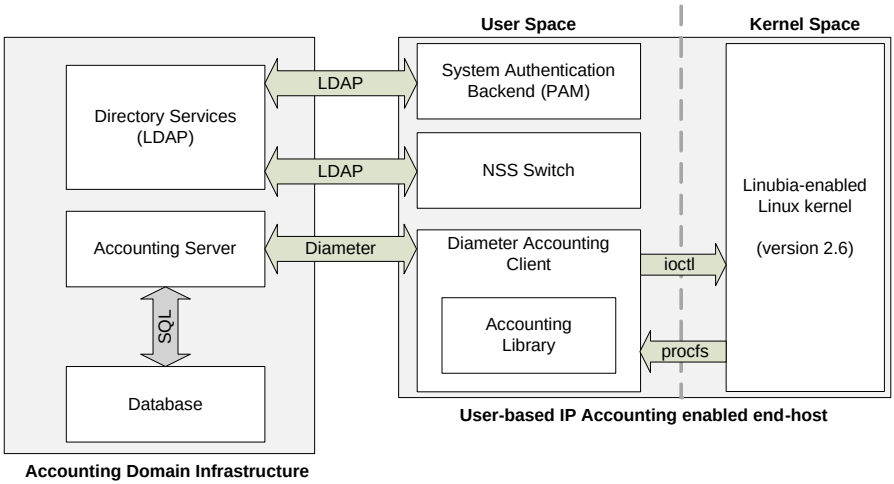


Figure 6.5: Integration of Components

- A protocol for transporting accounting information from the end nodes (such as Radius, Diameter, IPFIX)

For the design of a Linubia prototype, as well as for its implementation, a network architecture with LDAP as an authentication and authorization protocol, and Diameter as an accounting protocol, was chosen. The choice of the authentication, authorization, and accounting protocol does not influence the behavior of Linubia, so they can be replaced with any other protocol which provides the same functionality. The LDAP protocol allows the end-host system authentication backend to verify user credentials against a directory service. After the autenthication the Name Server Switch (NSS) uses the same LDAP protocol to retrieve additional relevant information (such as configuration files, group membership, access restrictions) for a user. An NSS-based configuration allows that a unique user account of a centralized user database (on a remote directory) can be used on any user host. The purpose of using such a centralized system is that multiple hosts use the same user database and therefore the same UIDs for individual users, making users and associated accounting records uniquely identifiable across distinct hosts.

The right hand side of Figure 6.5 shows the components required in the end host running Linubia. In addition to the kernel-specific accounting architecture sketched in Figure 6.3, two additional components are required for

building an accounting application on top of Linubia. The first component is an *accounting library* which describes an API for querying and configuring the accounting module. This library enables external applications to access the interfaces of the accounting extension in the kernel. The library sends information to the kernel module using the *ioctl* kernel interface, while the accounting data is read from the kernel module using the *procfs* file system. The second component required to build an accounting application ontop of Linubia is a Diameter accounting client that uses the accounting library to fetch the user-based IP accounting records from the kernel and sends them to a remote data aggregation server using the Diameter protocol. The aggregation server can evaluate and store the accounting data persistently, for example by using a separate database server.

6.3 Accounting Module Implementation

The implementation of the host-based extension follows an approach similar to the useripacct project [96] and is entirely written in the C programming language. LINUBIA supports 64 bit counters, provides real-time traffic statistics and allows parallel accounting of IPv4 as well as IPv6. The accounting system was implemented for modern 2.6 series Linux kernels and supports both IPv4 and IPv6. The implementation architecture is depicted in Figure 6.6. As the figure shows, Linubia includes a kernel module which is hooked to the kernel send and receive routines, and an access library (running in the user-space) which allows accounting data to be accessed by external applications.

The information triplet to be extracted from each IPv4 or IPv6 packet consists of the IP packet size, the packet owner (user), and the network and transport protocols involved with the operation. Unfortunately, the required routines and protocol headers are distinct for IPv4 and IPv6, and for incoming traffic, the information cannot be retrieved at the IP layer, like it is the case for outgoing traffic. This required the embedding of the accounting module routines in the transport layer implementation. A shortcoming of this approach is a scatter of the LINUBIA code across several files in the Linux kernel network subsystem. Figure 6.6 uses two abstractions for the sending and receiving routines called `recv_data()` and `send_data()` regardless of the sending or receiving protocols. Table 6.1 describes the actual names

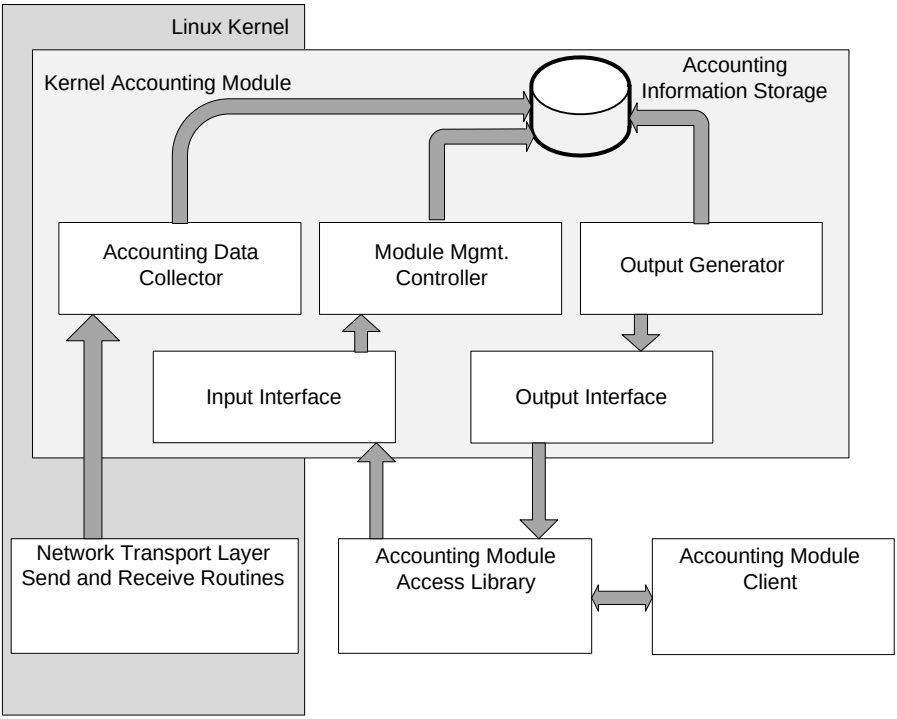


Figure 6.6: Implementation Architecture

and location of the sending and receiving routines for TCP, UDP, and ICMP in IPv4. Similarly, Table 6.2 provides the same information for IPv6.

Table 6.1: Location of send and receive routines for IPv4

Protocol	Source code	Routine name	Data structure
send TCP	/net/ipv4/ip_output.c	ip_queue_xmit()	struct sk_buff *skb
send UDP	/net/ipv4/ip_output.c	ip_push_pending_frames()	struct sock *sk
send ICMP	/net/ipv4/ip_output.c	ip_push_pending_frames()	struct sock *sk
recv RAW	/net/ipv4/raw.c	raw_rcv_skb()	struct sock * sk struct sk_buff *skb
recv UDP	/net/ipv4/udp.c	udp_rcv()	struct sk_buff *skb
recv TCP	/net/ipv4/tcp_ipv4.c	tcp_v4_rcv()	struct sk_buff *skb

The implementation architecture figure also outlines the interactions between the different components of LINUBIA. A system call from the Linux kernel to the sending or receiving routine triggers a call to the respective function - `add_rcv_traffic_to_user()` for the received traffic and `add_rcv_traffic_to_user()` for the sent traffic - in the accounting module and passes the user to which the traffic is assigned to, the amount of data, and the transmission protocol. This information is retrieved from a socket buffer structure (`struct sk_buff*`) which contains information about the socket which was used to send or receive data and also the data that was sent or received. Using the socket information details about the owner (process id and user id) of the socket are inferred.

Before the accounting module is called, the size of the data sent or received is retrieved from the socket information. The network and transport protocol types are determined by identifying the called network routine, while the user information is learned by looking up the ownership properties of the network socket through which the traffic was sent or received.

Table 6.2: Location of send and receive routines for IPv6

Protocol	Source code	Routine name	Data structure
send TCP	/net/ipv6/ip6_output.c	ip6_xmit()	struct sock *sk, struct sk_buff *skb,
send UDP	/net/ipv6/ip6_output.c	ip6_push_pending_frames()	struct sock *sk
send ICMP	/net/ipv6/ip6_output.c	ip6_push_pending_frames()	struct sock *sk
recv TCP	/net/ipv6/tcp_ipv6.c	tcp_v6_rcv()	struct sk_buff **pskb
recv UDP	/net/ipv6/udp.c	udpv6_rcv()	struct sk_buff **pskb
recv ICMP	/net/ipv6/raw.c	rawv6_rcv_skb()	struct sock * sk, struct sk_buff * skb

6.3.1 Accounting Data Collector

The accounting data collector exports two functions, which are used to inform the accounting module about new traffic sent or received by a user. These two functions have their signatures described in Table 6.3.

In order to verify that the user id received is valid, the function `ip_ipacct_find_user()` is called. This function returns NULL if a user does not exist, or a valid pointer if the user id is found. If the returned value is a valid pointer then the new traffic information is passed to the module controller which adds it to the storage component.

There are cases in which an incoming or outgoing IP packet cannot be mapped to a socket, and therefore not to a system user, either. Different tests have shown that the following situations lead to such situations:

- Premature destruction of a network socket: If an application sends a request to a remote host and closes the socket before having received a response, the incoming response packet(s) cannot be delivered to the socket (that doesn't exist any longer), and no user can be brought into relation to the incoming packets any longer.
- Access to non-existent services/sockets: If another machine tries to access a service on the local machine (e.g. makes a DNS request to a machine) incoming IP traffic is generated. Because it is not defined where this traffic belongs to, the packet is dropped and eventually an ICMP answer (service unreachable) is sent, thus the traffic cannot be credited to a regular system user.

Table 6.3: Accounting module sending and receiving routines

```
int ip6_acct_user_received
(uid_t user, long long recv, int l4proto );
int ip6_acct_user_sent
(uid_t user, long long sent, int l4proto);
```

IP traffic that cannot be accountable to a specific user is instead accounted to a special user (e.g. the user nobody, or root).

6.3.2 Accounting Information Storage

The *information storage* component is implemented as a number of records that are connected in groups of doubly-linked lists within a hash table. Each record contains the UID as the primary identification attribute as well as the measured IP traffic values for different network and transport protocols. Users are dynamically added when they start using IP networking. The

data structure used to store IP metering data for an user is shown in Table 6.4.

6.3.3 Module Management Controller

The module management controller component includes a set of routines to add an user to the storage component and to reset the traffic accounting data for a specific user inside the storage component.

The accounting data reset functionality requires the implementation of an input interface from user's space to kernel space. This is done by extending the socket-level I\/O control calls `inet_ioctls`. These are kernel functions that can be used from userspace (like Linux system calls). In this way it is possible to pass information from userspace to kernel space.

Table 6.4: Data structure for storing user traffic information

```
struct ip_acct_user_s
{
uid_tuid;
long longip4_udp_sent;
long longip4_tcp_sent;
long longip4_raw_sent;
long longip4_udp_recv;
long longip4_tcp_recv;
long longip4_raw_recv;
long longip6_udp_sent;
long longip6_tcp_sent;
long longip6_raw_sent;
long longip6_udp_recv;
long longip6_tcp_recv;
long longip6_raw_recv;
struct ip_acct_user_s *prev, *succ;
};
```

6.3.4 Output Generator

The *output generator* creates and maintains a `procfs` entry which contains IP metering data for all active users. Upon request it loops through the

user list and creates a table with all users and their traffic records for all protocols which is exported to the respective `procfs` entry. The user space library can read the `procfs` entry and retrieve from there the accounting data for each user.

6.3.5 Accounting Library

The accounting library recreates accounting record structures from metering data retrieved from the `procfs` file system so these can be easily accessed by other applications, such as the accounting client. It also provides functions to send commands to the *module controller*, using the `ioctl` interface.

Table 6.5: New AVPs for Linux User-Based IP

AVP Name	AVP Code	AVP Name	AVP Code
Linux-Input-IPV4-Octets	5001	Linux-Input-IPV4-TCP-Octets	5101
Linux-Output-IPV4-Octets	5002	Linux-Output-IPV4-TCP-Octets	5102
Linux-Input-IPV6-Octets	5003	Linux-Input-IPV4-UDP-Octets	5103
Linux-Output-IPV6-Octets	5004	Linux-Output-IPV4-UDP-Octets	5104
Linux-Input-TCP-Octets	5005	Linux-Input-IPV6-TCP-Octets	5105
Linux-Output-TCP-Octets	5006	Linux-Output-IPV6-TCP-Octets	5106
Linux-Input-UDP-Octets	5007	Linux-Input-IPV6-UDP-Octets	5107
Linux-Output-UDP-Octets	5008	Linux-Output-IPV6-UDP-Octets	5108

6.3.6 Accounting Client

The accounting client is a process running as a daemon on each end host. When started, the accounting client connects to an accounting server and starts collecting IP metering data for active users using the accounting library. Once a new active user is detected an accounting session is started for that user and the accounting client regularly sends accounting records to the accounting server for as long as that user remains active.

As already mentioned the accounting protocol used is Diameter. Within Diameter, records are structured as sets of (predefined) Attribute-Value Pairs (AVP). A set of AVPs is proposed by the Diameter protocol standard [14], but in order to support more granular accounting data (such as differentiation between TCP and UDP traffic) a set of parameters have been defined as shown in Table 6.5.

6.4 Chapter Summary

Linubia is another mechanism embedded in the distributed metering layer of DITA (c.f. Figure 3.3). It demonstrates by design and prototypical implementation that a user-based IP accounting approach is technically possible on modern Linux (2.6 series) operating systems. It can be used with both, IPv4 and IPv6 network protocols and it can be integrated into an existing accounting infrastructures, such as Diameter. The current implementation shows a clear proof of concept. Compared to traditional host-based accounting mechanisms, a user-based approach allows the mapping of network services usage not just to a device, but more specific, to the user which consumed those services. By the linkage of the networking subsystem to the socket interface, which also implies a link to the process management of the operating system, an advanced accounting module can offer IP accounting not only per user, but also per process. This allows for the identification, the management, or schedulability of processes not only by their CPU usage or memory consumption, but also by their network resource consumption. Finally, this leads to the creation of smart network filters or firewalls that allow for or deny network access to specific applications or users running on a host, instead of only allowing or denying specific services.

Chapter 7

Evaluation

In order to verify the viability of the proposed architecture, and dedicated mechanisms developed in this thesis, and to test their performance, a set of evaluation tests have been carried out in order to verify if the requirements set in Section 3.3 were met. The chapter starts by investigating the technical feasibility, and functionality of the developed mechanisms. The feasibility evaluation investigates the technical effort required to set-up and operate each of the three different mechanisms proposed by this thesis. Following, the performance of the implemented prototypes is evaluated and analyzed further in this chapter. Tests are performed which measure the overhead as well as the scalability of the proposed mechanisms. In addition, an overall analysis of a system integrating all three mechanisms is presented. The chapter concludes with a summary of the evaluation of this thesis in the context of the requirements established in Section 3.3.

The technical feasibility evaluation compares the requirements to deploy and operate each new mechanism with the typical resources and infrastructure available for a network operator. The evaluation shows that all the mechanisms are based on standardized protocol and can be easily integrated with an already existing traffic monitoring and analysis infrastructure. A performance evaluation is individually performed for each of the three components of this thesis (and their individual mechanisms), showing the scalability of each of those mechanisms as well as discussing possible disadvantages they introduce.

7.1 Technical Feasibility

The first part of the evaluation deals with an analysis of the applicability of the newly developed mechanisms into a real world scenario. The requirements in terms of hardware, software libraries, and other infrastructure elements are considered as well. As it can be observed in the following sections, none of the developed mechanisms require a technology which is not already available in the infrastructure of a network operator. Thus, the proposed mechanisms are feasible to be deployed in future network management systems. Moreover, the developed mechanisms are designed to be used with commodity hardware, which leads to an inexpensive deployment of the proposed solutions.

7.1.1 SCRIPT

In order to deploy a distributed SCRIPT application, a network operator needs to fulfill the following requirements:

1: metering data should be exported as IPFIX records

2: the deployment nodes should run Linux

3: the deployment nodes should be connected over an IP network

4: each deployment node should have SCTP enabled

5: the following libraries should be installed:
libsctp
libACE

Exporting data using the IPFIX protocol in future is considered as a technically feasible assumption, as it is the IETF standard for carrying IP flow information, and it was already proposed as a transport protocol for other type of metering data (*e.g.* by the PSAMP working group). Due to the use of templates, the protocol is flexible enough to carry virtually any type of data, so adapters that transform non-IPFIX data into IPFIX records can be easily implemented for already existing metering components that do not export IPFIX data (such as SNMP agents, or the Linubia user-based IP accounting module).

In order to deploy the implemented SCRIPT prototype, the SCRIPT nodes should run the Linux operating system. Most network operators have an important part of their network services, including network monitoring and management services, running on top of the Linux operating system, so they already have the knowledge and expertise for managing Linux machines. Even more, being an open source operating system, a distributed platform based on Linux would cost significantly less compared to a solution based on a commercial operating system. Thus, it is safe to consider the choice of the Linux operating system as a feasible requirement.

The third requirement is also assumed to be easy to fulfill, as it is improbable that a network operator cannot provide network connectivity to elements of its own infrastructure. However, some network operators might take special measures with respect to network connectivity, in order to improve transmission security, or to reduce the impact of IPFIX traffic on the productive traffic flowing through its links. Such measures include a dedicated VPN (Virtual Private Network) for the SCRIPT nodes, or even a dedicated network link (physical or logical).

Enabling SCTP on a Linux machine is straight forward, as most of the distributions nowadays come with a kernel which already has an SCTP module compiled. In the worst case a re-compilation of the Linux kernel to include the SCTP module is required. In case this requirement cannot be fulfilled, the SCRIPT prototype can also be used over UDP, but reliability of record delivery is not guaranteed in this case.

The implementation of the SCRIPT prototype is based on the libsctp and libace libraries, so an installation of these on each SCRIPT node prior to de-

ploying a SCRIPT application is required. As these are widely used libraries and because they are available in many different packaging formats for many Linux distributions, this is also considered as an easy to fulfill requirement.

As it is showed, SCRIPT is based on technology which is either already existing, either easy to add in a network operator's infrastructure. It is based on open standards, thus it can be integrated with other tools or equipment used by the network operator.

7.1.2 DiCAP

Similarly to SCRIPT, DiCAP was also designed with the purpose of being easily usable. The implemented prototype was built for the Linux 2.6 kernel. As DiCAP operates at a very low level, a slight modification of the network interface driver is required for the interfaces that receive the traffic to be monitored. The implemented prototype was built for a system with a NetXtreme BCM5721 1 Gigabit Ethernet card, but the changes in the network driver are easy to integrate for other network cards. The most important and technically challenging requirement a network operator needs to fulfill is to deliver to each DiCAP node a copy of the traffic to be monitored. For the evaluation of DiCAP a switch with port mirroring was used for this purpose and the traffic on the interface to be monitored was mirrored on several other interfaces. As the interfaces receiving mirrored traffic are only used for receiving traffic and they don't have to send any packets, inline network taps could be used to mirror traffic to all DiCAP nodes.

7.1.3 LINUBIA

LINUBIA also uses a kernel module for mapping IP packets to individual users, thus for each end-device on which LINUBIA needs to be deployed, the kernel needs to be updated in order to include this module. The authentication and authorization services do not directly interact with LINUBIA, and they are only required to make sure that the same user has the same user id on all machines. The accounting service may use a *pull* approach to get the IP metering data from the accounting module using the provided accounting library. As a result, the accounting module can be used with any authentication, authorization, and accounting system. LINUBIA can also be used with-

out a AAA infrastructure, but in this case IP metering data collected from several machines cannot be correlated based on the user id, and needs an external application to correlate these data.

7.2 Functional and Performance Evaluation

As the technical feasibility of the solution developed in this thesis has been shown, the next step is to perform a functional and performance evaluation of the three proposed mechanisms. The functional evaluation compares the achieved functionality with the requirements set in Section 3.3, while the performance evaluation tries to assess their benefit and overhead.

7.2.1 SCRIPT

The SCRIPT performance begins with an evaluation of three well-known hash functions in order to find the best choice for a hash function for routing IPFIX records. Following, an evaluation of the implemented prototype in a real deployment follows. The main purpose of SCRIPT is, to recall its major benefit, to distribute IPFIX records to several machines according to rules required by an analysis application. This is achieved by organizing participating nodes in a P2P overlay and by using the P2P overlay information for distributing the IPFIX records. Using the API provided, applications can define their specific IPFIX record routing rules according to their dedicated requirements. An application routing policy is specified as a hash function applied on a set of fields of an IPFIX record. The choice of the hash function has a high impact of the performance of a SCRIPT application, as it defines how the records are distributed. An ideal routing policy evenly distributes the IPFIX records to all participating SCRIPT nodes. The processing requirements of the hash function need also to be considered, as in case of high rate of IPFIX records the hashing should not be a bottleneck.

An extensive evaluation of hash functions has been performed in [51]. The authors compare a set of hash function with respect to their speed and hash distribution. The evaluation shows that the BOB hash function achieves the best choice to use when hashing IP content, as it achieves a good hash distribution with a very low time overhead. These results have been achieved using synthetically generated IP packets. Based on these results, the BOB

hashing function has been selected for SCRIPT. In order to validate the choice, a set of tests have been made in order to check how well BOB performs in case of real IP metering data, compared with other widely used hash functions.

Three hash functions have been further evaluated as possible choices for SCRIPT. These are BOB, SHA1 (Secure Hash Algorithm) [46], and MD5 (Message Digest 5 Algorithm) [85]. Three different data sets from three different networks have been used to evaluate these three hash functions.

The main differences compared to the evaluation of hash functions performed in [51] are:

- hash functions are evaluated on real traffic
- testing includes three data sets from different networks, showing different asymmetries in traffic
- several different field sets are used as input for the hash functions

Table 7.1: SCRIPT Data Sets

Data set source	Number of flow records	Observations
2 PlanetLab nodes	20.147.322	Highly asymmetric traffic with only two different PlanetLab nodes
University border router	31.035.415	Traffic to and from University of Twente (UT)
SWITCH backbone routers	34.184.342	Traffic aggregated from several universities in Switzerland

Three different data sets have been used to evaluate the performance of the three hash functions: one taken from the core network of a large operator, a second one consisting of flow records collected from the boarder router of a university network, and flow records collected from the link to two PlanetLab nodes. The first data set represents typical traffic that a larger network operator would see in its network, the second data set represents a traffic trace specific for an enterprise network, while the third data represents a highly asymmetric traffic between two IP addresses on one side and a high number of different addresses on the other side. The purpose of using these

three different data sets was to include a wider range of traffic characteristics, from traffic aggregated from multiple large institutions to traffic specific to a single lab.

Figure 7.1 shows the results of a test comparing BOB, MD5, and SHA hashing function with respect to the time they require to calculate a hash value. The test was performed on 3 million flow records collected in a real network, and each result represents the average of 3 million measurements. Four different input sets have been used for this test. The first input set consisted of the pair of port numbers of a flow record, which has 32 bits. The second input set includes the source IP address and source port number in each flow record. The third input set is 64 bits long and consists of the source and destination IP addresses. The fourth input set consists of source and destination IP addresses and port numbers and is 96 bits long. As Figure 7.1 shows the BOB hash function outperforms MD5 and SHA by running almost twice as fast as the second best choice. Another interesting result visible in the figure is the variation of hash calculation time with respect to the input size. It can be seen that in case of BOB a 200% increase in input length only requires about 15% more time. The figure does not show error bars, as the numbers are averages of 3 million measurements. .

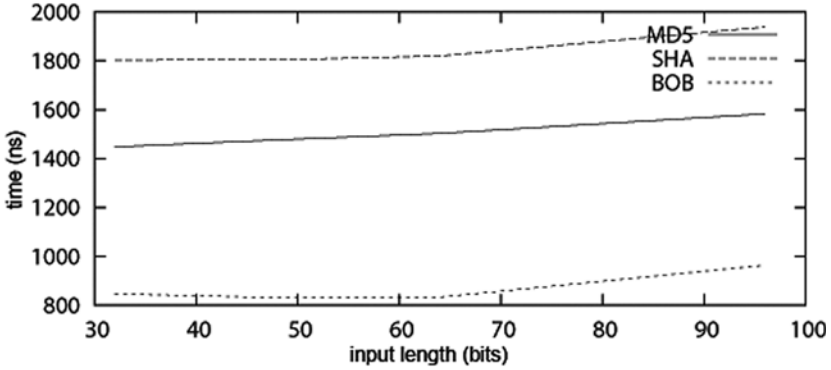


Figure 7.1: Hash computing time comparison

Table 7.2 shows a comparison of the three hash function with respect to how the hash values they generate on the three data sets are distributed to a set of 256 SCRIPT nodes. During the test four different sets of fields of a flow record have been used to calculate hash values. The numbers in the ta-

ble represent the standard deviation from a uniform distribution for the generated hash values. As the results show all three hash functions distribute the hash values very well, the maximum standard deviation in all tests being 2.76%. Another result of the test is the observation that all four sets of fields chosen for hash calculation can be used for routing in SCRIPT, as all four equally distribute the hash values to all SCRIPT nodes

Table 7.2: Hash Distribution

Fields	Hash Function	PlanetLab	SWITCH	UT
entire record	MD5	0.0821	0.0148	0.1202
	BOB	0.0822	0.0149	0.1201
	SHA	0.0822	0.0149	0.1202
ip_src, ip_dst	MD5	0.7356	0.8292	2.1203
	BOB	2.2968	2.1213	2.6126
	SHA	2.2976	2.2209	2.6139
port_src, port_dst	MD5	1.5553	1.5161	2.7672
	BOB	1.5545	1.5142	2.7666
	SHA	1.5553	1.5147	2.7664
ip_src, port_src	MD5	0.7356	0.8292	2.1203
	BOB	2.2968	2.2209	2.6126
	SHA	2.2976	2.2213	2.6139

As a SCRIPT network grows, the expected number of hops on the path of a flow record increases as well. Figure 7.2 shows the result of a theoretical calculation which measures the overhead introduced by SCRIPT. For this test different SCRIPT networks with sizes between 4 and 200 nodes have been simulated. For each of these networks it was calculated the total number of flow records transported by the SCRIPT network in order to deliver 1.000.000 flow records from NetFlow v9 exporters to SCRIPT nodes. As it can be seen in the figure, with 4 SCRIPT nodes, the number of flow records transported by SCRIPT is almost double the number of exported flow records. This happens because for 4 SCRIPT nodes, on average 75% of flow

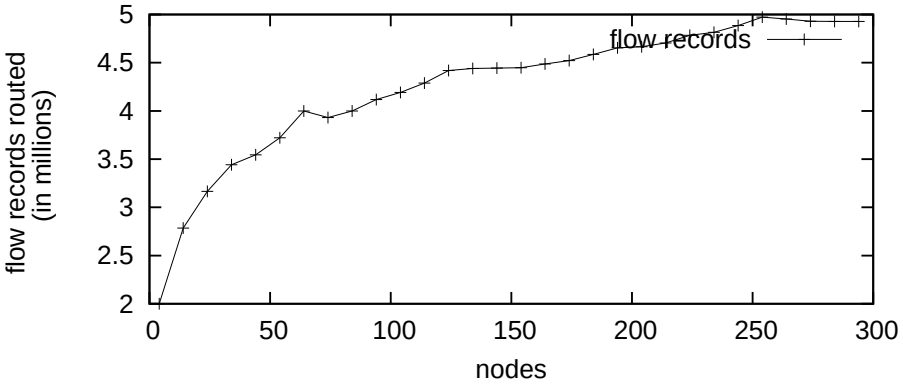


Figure 7.2: SCRIPT Overhead

records need to be routed. As the network is small, there is at maximum one intermediate SCRIPT node that has to perform routing. It can be seen in the figure that as the number of SCRIPT nodes grows, the overhead increase is logarithmic.

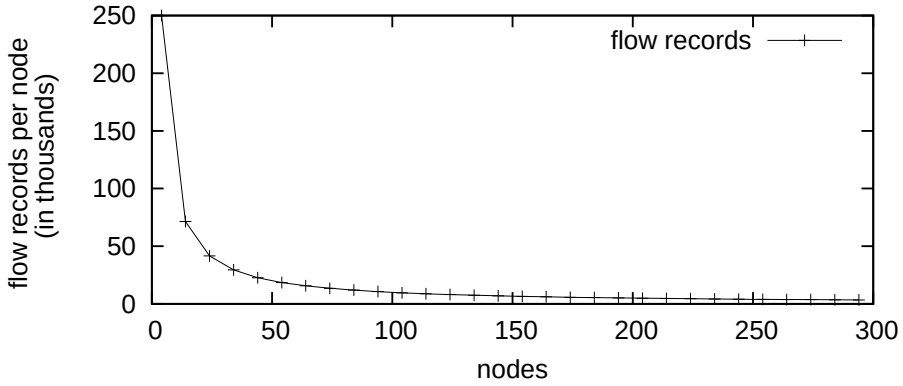


Figure 7.3: Flow Records per Node

In addition to the previous test, Figure 7.3 shows the number of flow records processed by a single node depending on the size of the SCRIPT network. The test assumes that 1.000.000 flows per second are sent to the SCRIPT network. The figure shows that the number of flow records pro-

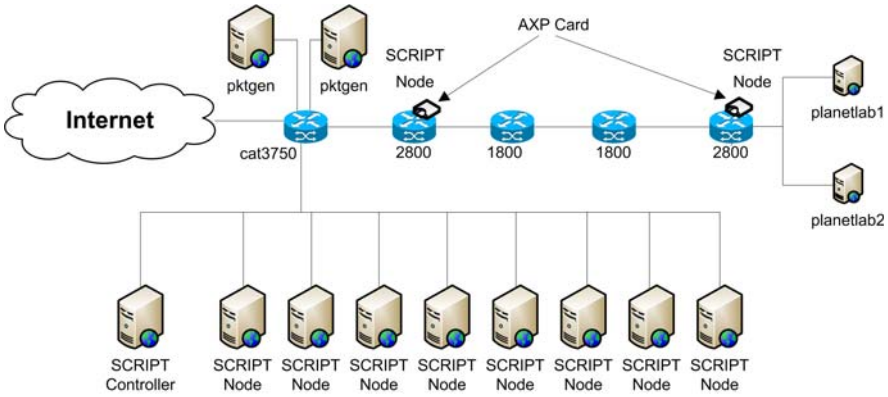


Figure 7.4: SCRIPT Evaluation Topology

cessed by a single node decreases proportionally with the number of nodes in the SCRIPT network.

As the above results only evaluate the performance of hash calculation, a set of tests have been performed in order to verify the behavior of SCRIPT in a real deployment. The topology of the deployed SCRIPT network is shown in Figure 7.4. A SCRIPT network was built using 8 SCRIPT nodes which were connected in the same LAN at 1 Gbps each. Two Linux machines were used to generate traffic with different characteristics. The *pktgen* tool available in the Linux kernel was used for traffic generation. All generated traffic was directed towards two PlanetLab machines (planetlab1 and planetlab2) via a set of intermediate routers. Two of these routers also hosted AXP cards which were used in some of the tests.

The first evaluation test on a deployment of a SCRIPT network checks if the collected flow records are distributed equally between processing nodes. The results of this test are shown in Figure 7.5 which outlines an average rate of flow records during a 60 seconds test using 8 SCRIPT nodes of about 62,000. As it can be observed in Figure 7.5, the maximum flow rate was 65,780 flows per second, while the minimum rate was at 60,535, resulting in a maximal deviation of 5% from the theoretical mean value.

The performance of the SCRIPT prototype is difficult to be assessed, especially in comparison with other tools, since no such generic frameworks for distributed IP metering data analysis exist. Therefore, the performance

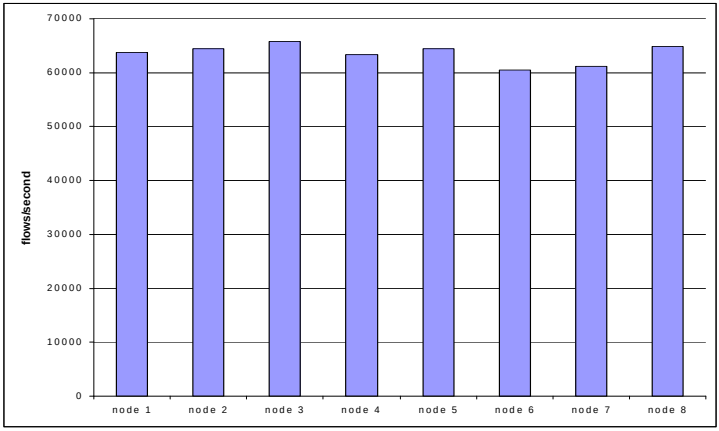


Figure 7.5: Distribution of Flow Records in SCRIPT

evaluation includes an evaluation of IPFIX records storage in a traditional, centralized collector, compared to the performance of a distributed collector built on top of SCRIPT. The tests were made using similar PCs with 3.6 GHz Intel processors, each having 4 GB memory. On the centralized collector the maximum rate of flow records that could be saved was 250,000 flows per second. Using SCRIPT running on 8 similar PCs in parallel a rate of 600,000 flows per second was achieved. In this evaluation, one stream of 150,000 flows per second was sent to 4 of the 8 nodes. Using only 4 nodes with SCRIPT the maximum flow rate that could be achieved in this prototype was 269,000 flows per second. These results are summarized in Figure 7.6.

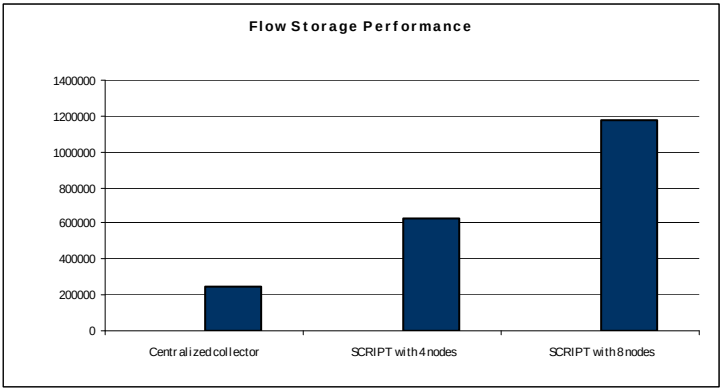


Figure 7.6: Flow Storage performance

During this evaluation it was observed that a single SCRIPT node can not process (in this case store in files) as many flow records as a similar centralized application running on the same node. The reason for this is that when running SCRIPT, some of the resources of a node are spent for calculating hash values and for the routing process, thus leaving less resources for the analysis application.

7.2.2 DiCAP

In order to evaluate DiCAP a set of performance tests have been performed. As the evaluation shows DiCAP offers a considerable performance improvement for Linux-based packet capture applications. The goal of the evaluation was to investigate what are the limitations of existing capturing tools (*libpcap* and *libpcap-PFRING*) and how DiCAP compares to them. Additionally DiCAP was used in *distributed analysis mode* as a distributed capture coordinator for *libpcap* and the combined performance of DiCAP and *libpcap* has been evaluated. The hardware used for evaluation consisted of single-core Pentium 4 nodes each running at 3.6 GHz, having each 1 GB RAM and each being connected to two Broadcom [12] Gigabit Ethernet network cards: one for receiving mirrored traffic and the other used for the communication with the coordinator node. The network traffic was generated using the Linux kernel packet generator *pktgen* [63]. In order to achieve the desired packet rates two nodes have been used to generate traffic. In addition, the packet rate was also controlled by modifying the packet sizes. The detailed evaluation setup is shown in Figure 7.7. The two packet generators were used to send data to one receiver in order to saturate a Gigabit Ethernet link. The link between the switch and the receiver was mirrored on five different ports towards four DiCAP nodes and one node running a libpcap application without DiCAP. A packet data analyzer was used by the DiCAP nodes to send observed packet headers during the tests with DiCAP nodes running in *distributed metering mode.*

The first test compares the capturing performance of *libpcap*, *libpcap-PFRING* and *DiCAP* on a single node at different packet rates. In the case of *libpcap* and *PFRING* a sample application was developed that counted the number of packets that were captured by those libraries. After each packet was captured and the counter incremented, the data was discarded. In this test, only one DiCAP node was used in *distributed metering mode.* For test-

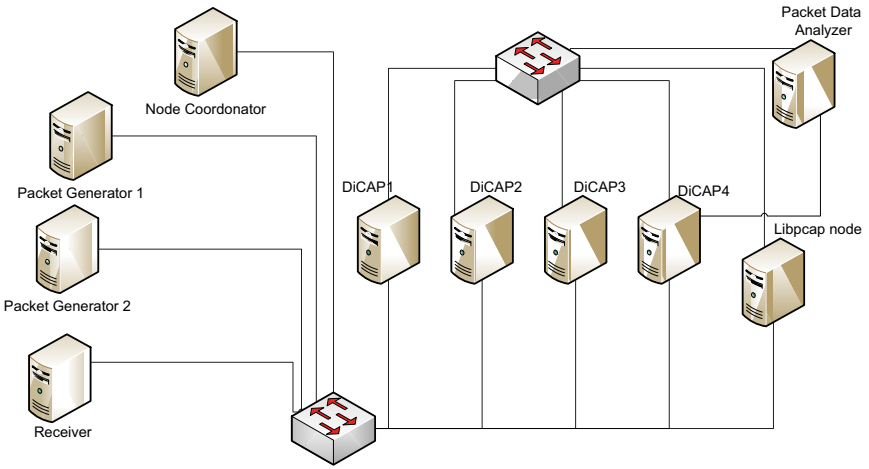


Figure 7.7: DiCAP Evaluation Setup

ing *DiCAP* the packet headers of each observed packet were sent to a *packet analyzer* node which counted the number of packet headers received. As the purpose of the test was to evaluate the performance on a single node, DiCAP was configured to capture every packet observed on the link by a single node. Table 7.3 shows that the performance of both *libpcap* and *libpcap-PFRING* is significantly affected by high packet rates. The loss rate in both cases was beyond 90% at 620.000 packets per second (pps). That rate was achieved by using 72 Byte packets which filled about 38% of the link capacity (376 Mbps). *DiCAP* experienced no loss at any packet rate showing it is the best solution out of the three evaluated for capturing packet headers at very high packet rates.

A second test was performed to check the empirical distribution of values in the identifier field of the IP packet header. As distribution of load in the capturing network is based on those values of the identifier field value it is important to check if those values follow a uniform discrete distribution. For this test traffic from a real network link was captured for two hours and a histogram was drawn. The total number of packets captured in that interval was 30 million packets. The result is shown in Figure 7.8. The result is slightly different than a roughly uniform distribution that was expected. The range of values for the identification field is 0-65355. The result of the test shows that two intervals cause the graphic not to show a uniform distribution function: 0

to 100 and 47.000 to 54.000. This behavior, most likely, is due to retransmissions or fragmentation of large transport layer payload.

Table 7.3: - Packet loss at high packet rates

Packet rate	libpcap loss	PFRING loss	DiCAP loss
119 Kpps	0%	0%	0%
232 Kpps	10%	1%	0%
380 Kpps	75%	28%	0%
492 Kpps	90%	83%	0%
620 Kpps	93%	96%	0%

Kpps = thousand packets per second

One conclusion that can be drawn from these results is that using the IP identification field in order to decide which packets to capture at each capture node, does not necessarily lead to a perfect load balance. However the approach is still acceptable as the total number of packets that were outside the uniform distribution function was below 5%. This observation does not impact other performance results of DiCAP, since the DiCAP prototype used for evaluation was based on the round-robin selection approach.

The third test investigates how *DiCAP* running in *distributed analysis mode* works in combination with *libpcap.* During this test the capturing capability of DiCAP has been disabled. Instead of capturing the packet header in this test DiCAP lets the selected packets pass through the kernel so that a libpcap application can capture them. The goal of this test is to evaluate the impact of using DiCAP with already existing libpcap applications. Three tests have been performed, each using different packet sizes: 40 byte, 256 byte, and 512 byte. The corresponding packet rates were: 625 Kpps, 480 Kpps, and 270 Kpps. Each test consisted of three parts. In the first measurement it was observed what percentage of the total number of packets sent by the packet generator are received by the *libpcap* application running on one capturing node with an unmodified Linux kernel. In the second measurement a setup with two capturing nodes, each running DiCAP and libpcap was prepared and the percentage of the captured packets was recorded. The third

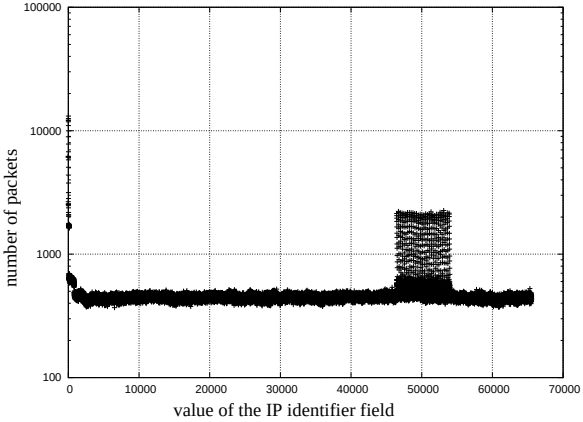


Figure 7.8: IP Identifier Field Value Histogram

measurement was performed similarly to the second one, but with four capture nodes instead of two. The results of the test are shown in Figure 7.9.

The figure shows that with traditional libpcap the maximum number of packets which are captured is around 50% for packet sizes of 512 Bytes. Lower loss rates were expected at larger packet sizes as the packet rate for these is lower. It can be observed that with two capture nodes running libpcap in parallel using DiCAP the capture rate was doubled for small packets and was increased more than 5 times for packet sizes of 256 Byte. With just two capture nodes it was possible to capture all packets of 512 Byte. With four capture nodes running in parallel libpcap the capture rate for very small packets (40 Byte) increased tenfold while for 256 Byte packets the capture rate was 100%. Another test not shown in the figure shows that with five parallel capture nodes a capture rate of 100% can be achieved even for the 40 Byte packet sizes.

As the first test shows, DiCAP can perform loss-less packet capture at high-speed packet rates using Linux. If one may say that the usability of the system in such a way is reduced as just the IP and transport header values are captured, the third test shows that DiCAP can be easily used with other capture tools in order to highly improve their performance. The main advantage of DiCAP can be seen not only in the observed performance boosts, but also

in its scalable architecture that allows for combined resources of multiple nodes to be used together more efficiently.

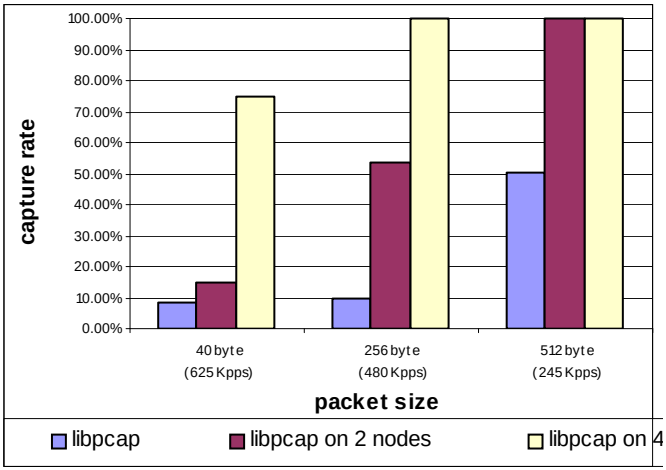


Figure 7.9: Performance improvement for libpcap applications

7.2.3 Linubia

The functional evaluation of the user-based IP accounting module first analyzed how well the requirements described in Section 3.3 have been met.

The set of experiments that have been performed in order to test the functionality, accuracy and performance of the accounting module used a network set-up as the one described in Figure 7.10. The testing environment consists of two hosts that are connected in a LAN by a Fast Ethernet switch as seen in the figure. Both hosts run a Linux 2.6 operating system and use IPv4 as well as IPv6. Both hosts have Fast Ethernet network adapters. All performance tests have been performed in a laboratory environment. For testing the functionality and robustness of the module LINUBIA was installed on an Ubuntu desktop machine and used in a production environment.

For testing the accuracy of the accounting module several tests have been performed in which TCP, UDP, and ICMP incoming and outgoing IPv4 and IPv6 traffic was generated and accounted for. The experiments have shown that the accounting module correctly accounts for IP traffic. During experi-

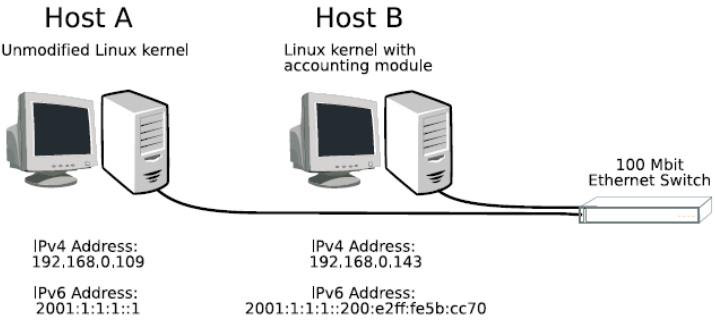


Figure 7.10: Linubia Testing environment

ments it was observed that some traffic cannot be mapped to any user (such as scanning traffic or incoming ICMP messages). Such traffic is accounted for the system user by the accounting module. Another observation concerns ICMP traffic that appears to be exclusively mapped to the system user and not to the user who actually sent the message. The reason for this is that raw socket operations are considered critical and only possible for user root, also for security reasons (a regular user can only execute the ping program because it has the *SUID-bit* set, thus being executed under root context).

Table 7.4 shows the results of a test consisting of a 256 MB file transfer over a Fast Ethernet link with and without LINUBIA using IPv4 and IPv6. The purpose of this test was to identify the impact of accounting on the performance of the Linux network subsystem. As the table shows there is only a small impact (0.83% for IPv4 and 0.41% for IPv6) on performance observed when running with LINUBIA enabled.

Table 7.4: Delay introduced by Linubia

	Unmodified IPv4	**Linubia IPv4**	**Unmodified IPv6**	**Linubia IPv6**
Average time	21.815 s	21.998 s	22.102 s	22.193 s
Std. deviation	0.062 s	0.208 s	0.010 s	0.204 s

In Table 7.5 observed and estimated maximum throughput on a Linux box with and without LINUBIA are shown. For estimating the maximum throughput the Iperf [53] tool was used. The test with Iperf affirms that the

measuring results are correct. Although the values are not totally equal, the dimensions are the same and the performance loss is marginal.

These tests have shown that LINUBIA delivers the required accounting results, while having a small impact on the performance of the end-system under investigation.

Table 7.5: Average maximum throughput

	Unmod. IPv4	Linubia IPv4	Rel. diff. (%)	Unmod. IPv6	Linubia IPv6	Rel. diff. (%)
Manual (Mbps)	93.880	93.099	0.839	92.661	92.281	0.412
Iperf (Mbps)	94.080	91.700	2.595	92.880	92.870	0.012

7.3 Evaluation Summary

The evaluation of the different mechanisms developed in this thesis show that distributed IP traffic analysis is a scalable solution to traffic metering and analysis in high speed networks. The evaluation shows (c.f. Table 7.6) that the requirements listed in Section 3.3 have been met. The approaches presented here are based on standardized protocols, so a seamless integration into existing network management systems is possible.

Table 7.6: Requirements match

RID	Requirement	How it is addressed
R1	*Scalable traffic analysis without sampling*	Scalability is addressed by increasing the computational resources and by reducing the network/storage/processing requirements of a a single analysis node.
R2	*Flexibility*	Analysis applications can be built ontop of the developed framework using the SCRIPT API.
R3	*Incremental scalability*	Increase in traffic to be analyzed is dealt with by adding new processing nodes.
R4	*High availability*	Nodes can join and leave. Offline nodes are detected and data cached until re-connection.

Table 7.6: Requirements match

RID	Requirement	How it is addressed
R5	*Superior price/performance*	The presented solution is based on inexpensive off-the-shelve PCs.
R6	*Ability to detect originating end-user or processes in case of network abuse*	Dedicated metering kernel module for Linux which does that.
R7	*Based on open standards*	As already shown, the solutions proposed here are based on standardized protocols, such as IPFIX, or Diameter, which are already used by network operators in their infrastructure.

By using incremental scalability, the amount of sampling used can be tuned according to the available resources which build the distributed analysis network, thus *R1* is fulfilled. Flexibility is given by the design of the distributed analysis network, which allows applications to be built ontop of it, and control the way IPFIX records are distributed. Incremental scalability is achieved by organizing the distributed analysis network into a self-organizing P2P overlay. Each time a node is added or removed from this overlay the distribution of work adapts to the new topology. As there is no single point of failure in the IPFIX distribution and processing path, the system still works, even if a node may go down. A possible single point of failure is the *CCR*, but it is not involved directly in the process or distribution of IPFIX records, but is only used when configuration needs to be passed to processing nodes. By using inexpensive off-the-shelve PCs, the distributed traffic analysis approach achieves better price/performance ration compared to a centralized state-of-the art supercomputer solution. By using Linubia it is also possible for administrators of multi-user systems to identify how much traffic a particular user, or process generated, without looking into the traffic which may be encrypted. In order to deploy the solution presented here into a network operator's premises no major changes need to be done, as the distribution of traffic data is based on IPFIX, and it also supports NetFlow v9 and v5.

Chapter 8

Conclusions and Future Work

Traffic metering and analysis still remain highly relevant topics in network management operations since most management operations depend on them. The year-to-year traffic increase trend observed during the last decade will most probably continue for the next years. Traditional centralized traffic metering and analysis solutions have shown scalability limitations as the amount of traffic to be analyzed increased. In order to cope with the challenges of traffic analysis of large networks today and in future, this thesis developed DITA, an architecture for distributed IP traffic analysis, which includes new mechanisms for sharing the workload of metering and analyzing IP traffic on high speed links between several devices. As their evaluation shows, these mechanism increase the performance of IP traffic metering and analysis by enabling multiple PCs to share their resources and distribute workload among them.

8.1 Conclusions per Research Problem

The conclusions for the three motivating questions introduced in Section 1.3 are shortly summarized below.

- *What is the effect of distributed IP traffic analysis.*

The investigations of this thesis show that by distributing the IP traffic metering data to multiple devices, a scalable infrastructure for IP traffic analysis may be built. Increase in IP traffic can be addressed by adding new devices to the analysis infrastructure. The SCRIPT framework was developed in this thesis to evaluate the new designed mechanisms. SCRIPT distributes flow records to multiple nodes and enables traffic analysis workload to be shared by multiple devices. Traffic analysis applications, like delay measurement or asymmetric route detection, access the SCRIPT functionality over a well-defined API. The SCRIPT framework uses a flexible routing function that can be specified according to the demands of each analysis application separately. It builds on standard protocols and supports IPFIX and NetFlow-based data transfer.

The SCRIPT framework has been implemented as a prototype and evaluated both on standard PC hardware as well as on Cisco AXP cards. The performance evaluations show that SCRIPT increases the total number of flow records processed compared to a centralized solution (*c.f.* Section 7.2.1 by ~500% with 8 SCRIPT nodes) and it scales with the total number of flow records exported in a network. The overhead introduced per SCRIPT node for flow record routing and relaying is low, so the Central Configuration Repository (CCR) does not introduce a new bottleneck. Since CCR is responsible only for management tasks and does not participate in the flow record transfer, is contacted rarely by SCRIPT nodes. As the evaluation reveals, the framework distributes flow records nearly equally among all nodes in the SCRIPT overlay, resulting in a fair balance of workload among all nodes.

- *How can the performance of traffic monitoring applications which run on off-the-shelve PCs be improved?*

The investigations of this thesis have shown that in case of software-based IP traffic monitoring, at high packet rates the performance bottleneck is the operating system reading the packets from the network interface card.

The approach investigated in this thesis is based on (a) cooperation between multiple PCs which receive each a copy of the traffic to be analyzed, and (b) a reduction of the number of packets the operating system needs to copy from the network interface card on every PC. Based on a cooperation protocol each PC can extract from the traffic a subset of packets which it analyzes, such that no packet will be investigated twice. A design and prototypical implementation for such a distributed packet capture mechanism named DiCAP was proposed. DiCAP does not require any dedicated hardware, which makes it a cost-effective solution for capturing IP packet headers at high packet rates using off-the-shelve PCs. As the evaluation results have shown in Section 7.2.2, when used in *distributed analysis mode* DiCAP achieves a performance increase of up to 500%, when two capture nodes were used in parallel, and up to 700%, when four capture nodes were used in parallel.

Being implemented as a LINUX open-source project, DiCAP can easily be extended with further functionality. The scalable architecture of DiCAP allows network professionals dealing with network measurements to increase strongly the performance of their measurements by adding new resources into the packet capturing infrastructure. The simplicity of the design allows DiCAP to be easily implemented in hardware leading towards a scalable hardware dedicated packet capture cluster architecture.

- *How to increase the granularity of IP metering data, so that an IP packet can be mapped to an individual user or even a process and application?*

As the link between an end-user (or a process) and an IP packet is lost once a packet leaves a network, the approach taken to account IP traffic on a per-user basis should make use of mechanisms embedded in end-devices, which can intercept network calls of processes an thus map each packet the the process which created (or received) the packet, or the user who owns that process. Linubia, the third mechanism developed in this thesis, shows by its design and prototypical implementation that such a user-based IP accounting approach is technically possible on modern Linux (running kernel version 2.6.x) operating systems. The design is IP protocol independent and can be used for IPv4 as well as for IPv6 traffic in parallel. Linubia's metering module can be easily integrated into an AAA infrastructure. The design presented shows a clear proof of concept which compared to traditional device-based accounting mechanisms allows the mapping of network traffic not only to a

device, but more specific, to the user which generated the respective traffic. Performing traffic metering at the linkage point between the networking subsystem and the socket interface allows accessing the process management structures of the operating system. Thus, new interesting mechanisms could be implemented, such as schedulability of processes based on network usage (besides the traditional CPU usage scheme). Linubia could also be used to create new network filters or firewalls that allow for or deny network access to specific applications or users running on a host, instead of only allowing or denying specific services. Additionally, new firewall and traffic scheduling policies could be designed so that a user might be blocked, or his traffic limited, once he exceeds some predefined traffic threshold.

8.2 Future Work

Future work should focus on integrating the current work on IPFIX mediation performed by the IETF in the results of this thesis. The result could be the starting point of a standardized distributed IPFIX mediation framework. In the context of the SCRIPT prototype, future work should focus on making SCRIPT more flexible by designing a mechanism that allows SCRIPT applications to be deployed ontop of a running SCRIPT network, without restarting the participant nodes. The current solution requires a recompile and redeployment of the system each time a change is made in an application, or a new application is deployed. A mechanism that allows applications to be added as *plugins* in a running SCRIPT network would significantly reduce both, the time required to deploy a SCRIPT application, and the unavailability of the other SCRIPT application due to service downtime.

Additionally, having such a *plugin* system available allows a larger scale of analysis questions to be formulated as small SCRIPT applications. The current approach is suited for long-lived traffic analysis applications that receives live traffic, while a flexible *plugin* based system would allow a query to be modelled as a SCRIPT application and historic traffic traces could be injected in this application. Finally, a built-in redundancy mechanism, independent of the application would increase the robustness of SCRIPT and would reduce the burden of an application developer in building a robustness mechanism in each SCRIPT application.

The developments of DiCAP could be used to create a scalable hardware-based traffic capture infrastructure. The mechanisms developed for the Linux kernel could easily be adapted and executed in hardware. Another extension to DiCAP could be the implementation of a filtering language which selects which traffic to be captured already at the DiCAP level, rather than at the application level. As currently DiCAP uses an own protocol to export packet headers when running in distributed metering mode, a useful addition would be the use of the IPFIX protocol to export metering data. Further improvement of DiCAP performance can be achieved by making use of the existing ring buffers on the network interface cards. Future work shall investigate how the integration of those ring buffers into the DiCAP architecture improve its performance. In addition, future work shall also concentrate on finding more suitable hashing functions for selection of packets.

The next obvious development with respect to Linubia is the implementation of per-process IP traffic accounting, as the current prototype only supports per-user accounting. The use of IPFIX to transport accounting data would also benefit Linubia, as this way the accounting data could be fed directly into SCRIPT for further processing.

8.3 Conclusion

This thesis investigated how the performance of IP traffic metering and analysis applications can be improved by switching from a centralized, high-performance infrastructure, which executes these tasks, to distributed mechanisms which combine the available resources of multiple devices. The results of these investigations show that distributed IP traffic metering and analysis leverages bottleneck problems which appear today in IP traffic metering and analysis. As with other proposals which address the challenges of analyzing IP traffic at high speeds (such as sampling, or aggregation), distributed IP traffic analysis does not solve all problems of handling such large amounts of data in very short time by itself, but proposes an orthogonal approach to existing solutions. It is the belief of the author of this thesis that combining distributed IP traffic metering and analysis with better and higher performance sampling and aggregation mechanisms may provide the solution to analyzing IP traffic in future high-speed networks.

As outcome of the these investigations, this thesis proposes a generic model for distributed traffic metering and analysis named DITA. In order to evaluate the model several different mechanisms have been developed and presented here, which show that traffic metering and analysis can scale with the increase of traffic carried by backbone network links. The need for such a solution is obvious, as most network operators already use sampling and aggregation techniques in order to reduce the amount of data they meter, in order to meet the capabilities of their existing metering and analysis infrastructure. The mechanisms developed during this thesis could bring benefits to network providers, service providers, as well as end users. By using SCRIPT and DiCAP, network providers can build a scalable infrastructure to meter and analyze their IP metering data, thus they have a more accurate view on the traffic they carry. As a result, service providers can be provided with better quality services, which finally increase the quality of service experience (QoE) for the end-users. By using Linubia, network administrators can have a more granular control and overview on the type and amount of traffic created or received by the users of their network.

All the mechanisms designed for DITA, and their prototypical implementations, are based on standard protocols and open-source technologies which make them easy to implement and integrate into existing infrastructures. To the knowledge of the author, this is the first approach to distributed IP traffic metering and analysis which (a) addresses the different bottlenecks of traffic analysis in a generic way, and (b) is self-organizing, thus offering a scalable solution to traffic increases.

Bibliography

[1] Adaptive Communication Environment Homepage: http://www.cs.wustl.edu/~schmidt/ACE.html, Last accessed: May 2010.

[2] A. O. Allen: *Probability, Statistics, and Queuing Theory with Computer Science Applications*, Academic Press, 1990.

[3] P.D. Amer, L.N. Cassel: *Management of sampled real-time network measurements*, 14th Conference on Local Computer Networks, Minneapolis, USA, 10-12 October 1989, pp. 62-68.

[4] K. G. Anagnostakis, M. Greenwald, R. S. Ryger: *Measuring Network-Internal Delays using only Existing Infrastructure*, 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2003, April 1-3, 2003, San Francisco, USA.

[5] M. D. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, D. Watson: *Data Reduction for the Scalable Automated Analysis of Distributed Darknet Traffic*, 5th ACM SIGCOMM Conference on Internet Measurement (IMC '05), pages 239-252, Berkeley, California, USA, October 2005.

[6] M. D. Bailey, E. Cooke, F. Jahanian, J. Nazario: *The Internet Motion Sensor - A Distributed Blackhole Monitoring System*, 12th Annual Network & Distributed System Security Symposium (NDSS '05), pages 167-179, San Diego, California, USA, February 2005.

[7] N. Benameur, J. Roberts:*Traffic Matrix Inference in IP Networks*, Networks and Spatial Economics, Vol.4 pp. 7-21, March 2004.

[8] Bittorrent official homepage: http://www.bittorrent.com/, Last accessed: May, 2010.

[9] Bob Jenkin, *BOB hashing function:* http://burtleburtle.net/bob/hash/doobs.html, Last accessed: September 2009.

[10] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, A. Lakhina, *Impact of Packet Sampling on Anomaly Detection Metrics*, 6th ACM SIGCOMM Conference on Internet Measurement, Rio de Janeiro, Brazil, October 25-17, 2006, pp 159-164.

[11] Bro IDS Homepage: http://www.bro-ids.org/, July 2008.

[12] Broadcom BCM 5721 Card, http://www.broadcom.com/products/Small-Medium-Business/Gigabit-Ethernet-Controllers/BCM5721, July 2008.

[13] N. Brownlee, C. Mills, G. Ruth: *Traffic Flow Measurement: Architecture*, IETF RFC 2722, October 1999.

[14] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, J. Arkko: *Diameter Base Protocol*, RFC 3588; September 2003.

[15] cflowd Project Webpage: http://www.caida.org/tools/measurement/cflowd/, Last accessed June 2010.

[16] C. Charnsripinyo, P. Roongroj: *Measuring the Internet Growth in Thailand*, International Symposium on Communications and Information Technologies, 2008. ISCIT 2008, , 21-23 Oct. 2008, Vientaine, Lao.

[17] Cisco Application Extension Platform: http://www.cisco.com/en/US/products/ps9701/, Last accessed: September 2009.

[18] Cisco Systems: *Approaching the Zettabyte Era*, White Paper, June 2008.

[19] Cisco Systems: *Hyperconnectivity and the Approaching Zettabyte Era*, White Paper, June 2009.

[20] Cisco Systems: *Introduction to Cisco IOS NetFlow - A Technical Overview*. White Paper, February 2006.

[21] K. C. Claffy, G. C. Polyzos, H. W. Braun: *Application of Sampling Methodologies to Network Traffic Characterization*, ACM SIGCOMM'93, San Francisco, California, U.S.A., September 13-17, 1993, pp 194-203.

[22] B. Claise: *Cisco Systems NetFlow Services Export Verrsion 9*, RFC 3954, October 2004.

[23] B. Claise: *Specification of the IP FLow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*, RFC 5101, January 2008.

[24] B. Claise, R. Wolter: *Network Management: Accounting and Performance Strategies*, Cisco Press, June 2007.

[25] B. Claise: *NetFlow/IPFIX Usage in Network Management*, EMANICS/IRTF-NMRG Workshop, Munchen, Germany, October 2008.

[26] W. Cochran: *Sampling Techniques*, Wiley, 1987.

[27] Cooperative Analysis of Internet Data: http://www.caida.org/, Last accessed: May 2010.

[28] J. Coppens, E.P. Markatos, J. Novotny, M. Polychronakis, V. Smotlacha, S. Ubik: *SCAMPI - A Scaleable Monitoring Platform for the Internet*, Proceedings of the 2nd International Workshop on Inter-Domain Performance and Simulation (IPS 2004), Budapest, Hungary, 22-23 March 2004.

[29] I. Cozzani, S. Giordano: *Traffic Sampling Methods for end-to-end QoS Evaluation in Large Heterogenous Networks*, Computer Networks and ISDN Systems, Vol. 30, Issue 16-18, September 1998.

[30] Data Retention Law in Germany: *Gesetz zur Neuregelung der Telekommunikationsüberwachung und anderer verdeckter Ermittlungsmaßnahmen sowie zur Umsetzung der Richtlinie 2006/24/EG* - http://www.rechtliches.de/info_Gesetz_zur_Neuregelung_der_Telekommunikationsueberwachung_und_anderer_verdeckter_Ermittlungsmassnahmen_sowie_zur_Umsetzung_der_Richtlinie_2006-24-EG.html, Last accessed: November 2010.

[31] Data Retention Law in Italy: Decreto-Legge n.144, 27 luglio 2005.

[32] L. Deri: *High-Speed Dynamic Packet Filtering*. Journal of Network and Systems Management, Vol. 15, No. 3, September 2007, pp 401-415.

[33] N.G. Duffield: *Sampling for Passive Internet Measurement: A Review*, Statistical Science,Vol. 19, No. 3, 472?498, 2004.

[34] N. G. Duffield, M. Grossglauser: *Trajectory Sampling for Direct Traffic Observation*, IEEE/ACM Transactions on Networking, Vol. 9, No. 3, June 2001, pp 280-292.

[35] N. G. Duffield, C. Lund, M. Thorup: *Learn More, Sample Less: Control of Volume and Variance in Network Measurement*, IEEE Transactions of Information Theory, Vol. 51, No. 5, pp. 1756-1775, May 2005.

[36] N.G. Duffield, C. Lund, M. Thorup: *Charging from Sampled Network Usage*, 1st ACM SIGCOMM Workshop on Internet Measurement, San Francisco, California, U.S.A., November 1-2, 2001, pp 245-256.

[37] N.G. Duffield, C. Lund, M. Thorup: *Properties and prediction of flow statistics from sampled packet streams*. ACM SIGCOMM Internet Measurement Workshop, November 2002, Marseille, France.

[38] N.G. Duffield, C. Lund: *Predicting resource usage and estimation accuracy in an IP flow measurement collection infrastructure.* ACM SIGCOMM Internet Measurement Conference, October 2003, Miami Beach, USA.

[39] N. G. Duffield, C. Lund, M. Thorup: *Flow sampling under hard resource constraints*. ACM SIGMETRICS 2004, pp. 85–96. ACM Press, New York.

[40] Endace Homepage: http://www.endace.com/, March 2008.

[41] C. Estan, G. Varghese: *New Directions in Traffic Measurement and Accounting*, ACM SIGCOMM Internet Measurement Workshop, San Francisco, California, U.S.A., November 2001, pp. 75-80.

[42] C. Estan, G. Varghese: *New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice*. ACM Trans. Comput. Syst. 21(3): 270-313 (2003).

[43] EU Parliament: *Directive 2006/24/EC of the European Parliament and of the Council on the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communications networks and amending Directive 2002/58/EC*, March 2006.

[44] M. Feier: *Design and Prototypical Implementation of a User-based IP Accounting Module for Linux*, Diploma Thesis, University of Zurich, February 2007.

[45] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee: *Hypertext Transfer Protocol -- HTTP/1.1*, IETF RFC 2616, June 1999.

[46] FIPS 180-2: Secure Hash Standard (SHS), National Institute of Standards and Technology, August 2002, ammended February 2004.

[47] flow-tools Project Homepage: http://www.splintered.net/sw/flow-tools/, Last accessed: April 2010.

[48] fProbe Project Site: http://sourceforge.net/projects/fprobe/, Last accessed: February 2009.

[49] S. H. Han, M.S. Kim, H. T. Ju, J. W. K. Hong: *The Architecture of NG-MON: A Passive Network Monitoring System for High-Speed IP Networks*, 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM), Montreal, Canada, October 21-23, 2002, pp 16-27.

[50] D. Harrington, R. Presuhn, B. Wijnen: *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*, RFC 3411, December 2002.

[51] C. Henke, C. Schmoll, T. Zseby: *Empirical evaluation of hash functions for multipoint measurements*, SIGCOMM Computer Communications Review Vol. 38, Nr. 3, July 2008, pp 39-50.

[52] IEEE IPFIX Working Group Homepage: http://www.ietf.org/ html.charters/ipfix-charter.html, April 2008.

[53] Iperf Homepage: http://dast.nlanr.net/Projects/Iperf/, May 2007.

[54] ISO/IEC Standard 10040: *Information technology - Open Systems Interconnection - Sytems management overview*, 1998.

[55] ITU-X Recommendation X.509 : *Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks*, 1988.

[56] Juniper Networks: *Configure traffic sampling*. Available at www.juniper.net/techpubs/ software/junos/junos63/swconfig63-policy/html/sampling-config4.html.

[57] Y. Kitatsuji, K. Yamazaki: *A Distributed Real-time Tool for IP-flow Measurement*, International Symposium on Applications and the Internet, Tokyo, Japan, January 26-30, 2004, pp 91-98.

[58] A. Kobayashi, B. Claise: *IPFIX Mediation: Problem Statement*, IETF Internet Draft, http://www.ietf.org/id/draft-ietf-ipfix-mediators-problem-statement-07.txt, December 2009.

[59] A. Kobayashi, B. Claise, K. Ishibashi: *IPFIX Mediation: Framework*, IETF Internet Draft, http://www.ietf.org/id/draft-ietf-ipfix-mediators-framework-04.txt, October 2009.

[60] S. Leinen: *Evaluation of Candidate Protocols for IP Flow Information Export*, RFC 3955, October 2004.

[61] libpcap Homepage: http://www.tcpdump.org, Last accessed: February 2009.

[62] libpcap-PFRING Homepage: http://www.ntop.org/PF_RING.html, Last accessed: February 2008.

[63] Linux Online Homepage: http://www.linux.org/, March 2008.

[64] J. Mai, C. Chuah, A. Sridharan, H. Z. T. Ye: *Is Sampled Data Sufficient for Anomaly Detection?*, 6th ACM SIGCOMM Conference on Internet Measurement, Rio de Janeiro, Brazil, October 25-17, 2006, pp 165-176.

[65] R. Mahajan, S. Floyd, D. Wetherall: *Controlling High-Bandwidth Flows at the Congested Routers*, 9th International Conference on Network Protocols, Riverside, California, US, November 2001.

[66] P. Maymounkov, D. Mazières: *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*, IPTPS 2002, March 2002, Cambridge, MA, USA.

[67] Y. Mao, K. Chen, D. Wang, W. Zheng: *Cluster-based Online Monitoring System of Web Traffic*, 3rd International Workshop on Web Information and Data Management, Atlanta, Georgia, U.S.A., November 9-10, 2001, pp. 47-53.

[68] K. McCloghrie, D. Perkins, J. Schoenwaelder: *Structure of Management Information Version 2 (SMIv2)*, IETF RFC 2578, April 1999.

[69] Minnesota Internet Traffic Studies (MINTS) Website: http://www.dtc.umn.edu/mints/home.php, Last accessed: February 2010.

[70] M. Molina: *A scalable and efficient methodology for flow monitoring in the Internet*, International Teletraffic Congress (ITC-18), Berlin, Sep. 2003.

[71] C. Morariu, P. Racz, B. Stiller: *SCRIPT: A Framework for Scalable Real-time IP Flow Record Analysis*, 12th IEEE/IFIP Network Operations and Management Symposium (NOMS 2010), IEEE, Osaka, Japan, April 2010.

[72] C. Morariu, P. Racz, B. Stiller: *Design and Implementation of a Distributed Platform for Sharing IP Flow Records*, 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2009), IEEE, Venice, Italy, October 2009.

[73] C. Morariu, B. Stiller: *DiCAP: Distributed Packet Capturing Architecture for High-Speed Network Links*, 33rd Annual IEEE Conference on Local Computer Networks (LCN), Montreal, Canada, October 2008.

[74] C. Morariu, Thierry Kramis, B. Stiller: *DIPStorage: Distributed Storage of IP Flow Records*, 16th IEEE Workshop on Local and Metropolitan Area Networks, Cluj-Napoca, Romania, September 2008.

[75] C. Morariu, Manuel Feier, B. Stiller: *LINUBIA: A Linux-supported User-Based IP Accounting*, 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2007), San Jose, USA, October 2007.

[76] C. Morariu, B. Stiller: A Distributed Architecture for IP Traffic Analysis, Inter-Domain Management, Autonomous Infrastructure, Management and Security, AIMS 2007, Oslo, Norway, June 2007, pp. 216-220.

[77] NfDump Project Homepage: http://nfdump.sourceforge.net, Last accessed: April 2010.

[78] NfSen Project Homepage: http://nfsen.sourceforge.net, Last accessed: April 2010.

[79] Ntop Project Homepage: http://www.ntop.org/, February 2008.

[80] A.M. Odlyzko: *Internet traffic growth: Sources and implications*, Proceedings of SPIE, Vol. 5247, August 2003, pp.1-15.

[81] D. A. Patterson, J. L. Hennessy: *Computer Organization and Design*, fourth ed., Morgan Kaufmann, 2008.

[82] J. Quittek, S. Bryant, B. Claise, P. Aitken, J. Meyer: *Information Model for IP Flow Information Export*, RFC 5102, January 2008.

[83] J. Quittek, T. Zseby, B. Claise, S. Zander: *Requirements for IP Flow Information Export*, RFC 3917, October 2004.

[84] P. Racz: *A Generic Accounting Configuration Architecture for Multi-Service Mobile Networks*, Doctoral Thesis, University of Zurich, ISBN 978-3-8322-7366-8, Shaker Verlag, Aachen, Germany, June 2008. [

[85] R. Rivest: *The MD5 Message-Digest Algorithm*, IETF RFC 1321, April 1992.

[86] L.G. Roberts: *Beyond Moore's law: Internet growth trends*, IEEE Computer Mag., January 2000.

[87] G. Sadasivan, N. Brownlee, B. Claise, J. Quittek: *Architecture for IP FLow Information Export*, March 2009.

[88] F. Schneider, J. Wallerich: *Performance Evaluation of Packet Cacpturing Systems for High-Speed Networks*, 2005 ACM conference on Emerging network experiment and technology, Toulouse, France, October 2005.

[89] J. Sermersheim: *Lightweight Directory Access Protocol (LDAP): The Protocol*, IETF RFC 4511, June 2006.

[90] Snort Homepage: http://www.snort.org/, July 2008.

[91] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, B. Stiller: *An Overview of IP Flow-based Intrusion Detection*, IEEE Communications Surverys & Tutorials, ISSN 1553-877X, IEEE, Vol. 12, No. 3, April 2010.

[92] G. Stampfel, W. Gansterer, M. Ilger: *Data Retention - The EU Directive 2006/24/EC from a Technological Perspective*, Medien und Recht Publishing, Wien-München, 2008.

[93] Tcpdump/ Repository Homepage, http://www.tcpdump.org/, February 2008.

[94] B. Trammell, E. Boschi: *Bidirectional Flow Export Using IP Flow Information Export (IPFIX)*, RFC 5103, January 2008.

[95] P. Tune, D. Veitch: *Towards optimal sampling for flow size estimation*. In Proceedings of the 8th ACM SIGCOMM Conference on internet Measurement (Vouliagmeni, Greece, October 20 - 22, 2008). IMC '08. ACM, New York, NY, 243-256.

[96] UserIPacct Homepage: http://ramses.smeyers.be/homepage/useripacct/, May 2007.

[97] W. Van Wanrooij, A. Pras: *Data on retention*, 16th IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management (DSOM), Barcelona, Spain, October 2005.

[98] Wolfgang J., S. Tafvelin: *Analysis of Internet Backbone Traffic and Header Anomalies Observed*, 7th ACM SIGCOMM Conference on Internet Measurement, San Diego, California, U.S.A., October 24-26, 2007, pp 111-116.

[99] Wuala Homepage: http://wua.la/, April 2008.

[100] Y. Xuebiao, L. Xiaodong, J. Jian, Y. Baoping: *Measuring Internet Growth from DNS Observations*, Second International Conference on Future Information Technology and Management Engineering, FITME '09, 13-14 Dec. 2009, Sanya, China.

[101] T. Zseby: *Statistical Sampling for Non-Intrusive Measurements in IP Networks*, Ph. D. Thesis, Technische Universität Berlin, Universitätsbibliothek (Diss.-Stelle), Fakultät IV — Elektrotechnik und Informatik, 2005.

[102] T. Zseby, E. Boschi, N. Brownlee, B. Claise: *IP Flow Information Export (IPFIX) Applicability*, RFC 5472, March 2009.

[103] T. Zseby, M. Molina, N. Duffield, S. Niccolini, F. Raspall: *Sampling and Filtering Techniques for IP Packet Selection*, IETF RFC 5475, March 2009.

Publications of the Author besides the Thesis Topic:

- C. Morariu, B. Stiller: *An Open Architecture for Distributed IP Traffic Analysis (DITA)*. 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), Dissertation Digest, IEEE, May 2011.
- B. Stiller, F. De Turck, C. Morariu, M. Waldburger: *Report on the 4th International Conference on Autonomous Infrastructures, Management, and Security (AIMS 2010) and the International Summer School on Network and Service Management (ISSNSM 2010)*. Journal on Networks and System Management (JNSM), ISSN 1064-7570, Springer, Vol. 19, No. 1, pages 130-136, January 2011.
- S. Kotrotsos, P. Racz, C. Morariu, K. Iskioupi, D. Hausheer, B. Stiller: *Business Models, Accounting and Billing Concepts in Grid-aware Networks*. Third International ICST Conference on Networks for Grid Applications (GridNets 2009), Springer, September 2009.
- B. Stiller, T. Bocek, F. Hecht, C. Morariu, P. Racz, A. Vancea, M. Waldburger (Eds.): *Communication Systems III*. IFI Technical Report, June 2009.
- B. Stiller, C. Morariu, P. Racz, M. Waldburger (Eds.): *Internet Economics IV*. IFI Technical Report, No. ifi-2009.01, February 2009.
- C. Morariu: *A Distributed Architecture for IP Traffic Analysis*. Joint EMANICS/IRTF-NMRG Workshop on Netflow/IPFIX Usage in Network Management, October 2008.
- F. Victora Hecht, T. Bocek, C. Morariu, D. Hausheer, B. Stiller: *LiveShift: Peer-to-peer Live Streaming with Distributed Time-Shifting*. Eighth IEEE International Conference on Peer-to-Peer Computing (IEEE P2P'08), Demo Session, September 2008.
- D. Hausheer, T. Bocek, F. Victora Hecht, C. Morariu, G. Schaffrath, B. Stiller (Eds.): *P2P Challenge Task 2008*. IFI Technical Report, July 2008.
- B. Stiller, T. Bocek, F. Hecht, C. Morariu, P. Racz, G. Schaffrath (Eds.): *Communication Systems II*. IFI Technical Report, No. ifi-2008.01, January 2008.
- B. Stiller, D. Hausheer, J. Gerke, P. Racz, C. Morariu, M. Waldburger: *Accounting and Charging - Guarantees and Contracts*. Ubiquitous Computing Technology for Real Time Enterprises, Max Mühlhäuser, Iryna Gurevych (Eds.), ISBN 978-1-59904-832-1, Information Science Reference, pages 302-336, December 2007.
- C. Morariu, P. Racz, D. Hausheer, B. Stiller: *Diameter Grid Accounting Application*. IETF Internet Draft, October 2007.

- T. Kirkham, D. Lutz, P. Mandic, J. Movilla, J. Gallop, C. Morariu: *Identity Management in a Mobile Grid Environment* . UK e-Science 2007 All Hands Conference, September 2007.
- P. Racz, J. E. Burgos, N. Inacio, C. Morariu, V. Olmedo, V. Villagra, R. L. Aguiar, B. Stiller: *Mobility and QoS Support for a Commercial Mobile Grid in Akogrimo*. 16th IST Mobile & Wireless Communications Summit, July 2007.
- B. Stiller, T. Bocek, Hasan, Pascal Kurtansky, C. Morariu, P. Racz, G. Schaffrath, M. Waldburger (Eds.): *Internet Economics III*. IFI Technical Report, No. ifi-2007.09, July 2007.
- D. Hausheer, T. Bocek, C. Morariu, G. Schaffrath, B. Stiller (Eds.): *P2P Challenge Task 2007*. IFI Technical Report, July 2007.
- Hasan, P. Racz, C. Morariu, D. Hausheer, B. Stiller: *A4C Support for Commercialization of Next Generation Grid Services*. ERCIM NEWS, No. 70, pages 18-19, July 2007.
- B. Stiller, T. Bocek, C. Morariu, P. Racz, G. Schaffrath, M. Waldburger (Eds.): *Mobile Systems II*. IFI Technical Report, No. ifi-2007.04, March 2007.
- M. Waldburger, C. Morariu, P. Racz, J. Jähnert, S. Wesner, B. Stiller: *Grids in a Mobile World: Akogrimo's Network and Business Views*. Praxis der Informationsverarbeitung und Kommunikation (PIK), Vol. 30, No. 1, pages 32-43, January 2007.
- C. Morariu, M. Waldburger, B. Stiller: A*n Integrated Accounting and Charging Architecture for Mobile Grids*. Third International Workshop on Networks for Grid Applications (GridNets 2006), pages 1-10, October 2006.
- C. Morariu, R. Zimmermann, B. Stiller: *Implementation of a Non-Repudiation Approach for Web-Services in Mobile Grids*. 12th Eunice Summer School 2006, September 2006.
- B. Stiller, T. Bocek, D. Hausheer, C. Morariu, P. Racz, G. Schaffrath, M. Waldburger: *Communication Systems I*. IFI Technical Report, August 2006.
- M. Waldburger, C. Morariu, P. Racz, J. Jähnert, S. Wesner, B. Stiller: *Grids in a Mobile World: Akogrimos Network and Business Views*. IFI Technical Report, April 2006.
- C. Morariu, M. Waldburger, B. Stiller: *An Accounting and Charging Architecture for Mobile Grids*. IFI Technical Report, April 2006.
- B. Stiller, T. Bocek, C. Morariu, P. Racz, M. Waldburger: *Internet Economics II*. IFI Technical Report, February 2006.
- B. Stiller, C. Morariu, P. Racz, M. Waldburger: *Mobile Systems I*. IFI Technical Report, July 2005.
- B. Stiller, C. Morariu, P. Racz, M. Waldburger: *Internet Economics I*. IFI Technical Report, March 2005.

List of Abbreviations

A

ACE	Adaptive Communication Environment
ACK	Acknowledgement
API	Application Programming Interface
APPID	SCRIPT Application ID
AS	Autonomous System
AVP	Attribute-Value-Pair
AXP	Application Extension Platform

C

CAIDA	Cooperative Association for Internet Data Analysis
CCR	Central Configuration Repository
CPU	Central Processing Unit

D

DHT	Distributed Hash Table
DITA	Distributed IP Traffic Analysis
DRAM	Dynamic Random Access Memory
DTLS	Datagram Transport Layer Security

G

GTID	Global Template Identifier

H

HTTP	Hypertext Transfer Protocol

I

ICMP	Internec Control Message Protocol
IDS	Intrusion Detetection System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocolversion 4
IPv6	Internet Protocol version 6
IPFIX	Intenet Flow Information Export
ISP	Internet Service Provider

K

Kpps	Thousands of packets per second

L

LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LRU	Least Recently Used

M

MAC	Medium Access Control
MD5	Message-Digest algorithm 5
MIB	Management Information Base
MPLS	Multi Protocol Label Switching

N

NG-MON	Next Generation Traffic Monitoring and Analysis System
NIC	Network Interface Card
NIDS	Network Intrusion Detection System

O

OC	Optical Carrier

P

P2P	Peer-to-Peer
PC	Personal Computer
POP	Point of Presence
PPU	Pre-processing Unit
PSAMP	Packet Sampling

Q

QoE	Quality of Experience
QoS	Quality of Service

R

RAM	Random Access Memory
RID	Routing Hash Identifier

S

SCTP	Stream Control Transport Protocol
SDK	Source Development Kit
SHA	Secure Hash Algorithm
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SRAM	Static Random Access Memory
SYN	Synchronize

T

TCP	Transport Control Protocol
TID	Template Identifier
TLS	Transport Layer Security

U

UDP	User Datagram Protocol

UID User Identifier

V

VoIP Voice over IP

List of Figures

List of Tables

Acknowledgements

I would like to thank to a few people which contributed to the completion of this work. First and foremost I would like to thank Prof. Dr. Burkhard Stiller for enabling and supporting the research presented in this thesis, as well as for his support during the different stages of my research which helped shaping this thesis. I would also like to thank my co-referent, Prof. Dr. Aiko Pras for his valuable feedback.

I would also like to thank Dr. Peter Racz for the lenghty talks about traffic accounting and analysis, for supporting this work, and for being a great office mate. Special thanks go to Benoit Claisse and Ralf Wolter from Cisco for their valuable input from an industry point of view.

I thank members of the Department of Informatics at University of Zurich for feedback and interesting discussions, especially (in alphabetical order) Thomas Bocek, Dr. Hasan, Dr. David Hausheer, Fabio Hecht, Guillerme Machado, Peter Ming, Andrei Vancea, Martin Waldburger. Also I thank the students which contributed to the work of this thesis, namely Nicolas Baumgardt, Manuel Feier, Thierry Kramis.

Finally I am ever thankful to my wife Lucia for her continuous love and support. Furthermore I would like to thank my parents for their dedication and support in all my actions.

Curriculum Vitae

Cristian Morariu was born in Cluj-Napoca, Romania, on September 20, 1980. Between 1999 - 2004 he studied at the Technical University of Cluj-Napoca where he graduated with a diploma in Computer Engineering. During his studies he worked as a system and network administrator at the Network Communication Center of the Technical University of Cluj-Napoca, and the Cluj-Napoca Network Operations Center of RoEduNet.

From 2004 to 2010 Cristian Morariu was a doctoral student and employed as a research assistant in the Communication Systems Group at the Department of Informatics of University of Zurich. During this time he participated in the European Union projects “Access to Knowledge through the Grid in a mobile World (Akogrimo)”, “Europe-China Grid InterNetworking (ECGIN)”, as well as in “European Network of Excellence for the Management of Internet Technologies and Complex Services (EMANICS)”.

His main research interests are IP traffic accounting and monitoring for high-speed networks. During his PhD thesis he was supervised by Prof. Dr. Burkhard Stiller.